END OF COURSE PROJECT (PFE) REPORT

May to October 2025

Dexterous objects manipulation learning for a humanoid robot hand

Ocean NOEL
FISE 25 – ROB

ENSTA Bretagne

2 rue francois verny,
Brest 29200,
France.

Supervisor: Gilles LE CHENADEC
gilles.le_chenadec@ensta.fr (Website)

Université Montpellier - CNRS LIRMM

UMR 5506 860 rue St Priest,
34095 Montpellier cedex 5, France.

Supervisor: Kheddar ABDERRAHMANE
Abderrahmane.Kheddar@lirmm.fr (Website)

Abstract (ENGLISH)

This report presents the work conducted during my internship at LIRMM in collaboration with Honda Research, in preparation for a future PhD. The objective was to investigate reinforcement learning approaches for dexterous object manipulation with a humanoid robot hand. After introducing the context, motivations, and challenges of this task, the report reviews the state of the art in both manipulation and continual learning, and details the chosen simulation framework, agents, and training setup. Experimental results are discussed, along with identified limitations, leading to the exploration of continual learning strategies and a novel approach. The report concludes with current findings and perspectives for future work.

Abstract (FRENCH)

Ce rapport présente le travail réalisé durant mon stage au LIRMM en collaboration avec Honda Research, en préparation d’une future thèse de doctorat. L’objectif était d’étudier des approches d’apprentissage par renforcement pour la manipulation d’objets avec une main robotique humanoïde. Après avoir introduit le contexte, les motivations et les défis liés à cette tâche, le rapport passe en revue l’état de l’art en matière de manipulation et d’apprentissage continu, puis décrit le cadre de simulation choisi, les agents utilisés et la configuration des entraînements. Les résultats expérimentaux sont présentés, ainsi que les limitations identifiées, menant à l’exploration de stratégies d’apprentissage continu et à la proposition d’une nouvelle approche. Le rapport se conclut par les enseignements tirés et les perspectives pour de futurs travaux.

1 Introduction

1.1 Internship context & Goals

This internship was carried out at the "Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier" (LIRMM) [**1**] in collaboration with Honda Research Japan [**2**], as part of the preparation for a future PhD project under the supervision of M. Abderrahmane Kheddar. LIRMM is a joint research unit of the University of Montpellier [**3**] and CNRS [**4**], specializing in computer science, robotics, and microelectronics, with a strong track record in both academic and industrial collaborations. M. Abderrahmane Kheddar is a CNRS research director of the Interactive Digital Humans (IDH) team [**5**], internationally recognized for his work in humanoid robotics, haptics, and human-robot interaction.

The internship focused on developing AI-based learning systems enabling a humanoid robot to skillfully manipulate multiple objects within its hand.
In particular, the internship aims to:

- Model and simulate object manipulation using advanced simulation environments for realistic contact-rich scenarios.
- Apply reinforcement learning (RL) [**6**] to train the robot to uses external forces and its own body for manipulation in diverse contexts.
- Integrate multimodal sensory feedback (vision, force/torque sensing, proprioception) to improve learning robustness and adaptability.
- Optimize learning strategies to improve sample efficiency and training time.
- Experiment on real hardware if simulation results are promising.

The internship was organized into three 2-months main phases, as shown in **Figure 1**. The first phase focused on configuring the simulation environment, defining tasks, and conducting a state-of-the-art review. The second phase centered on developing reinforcement learning-based architectures for skill acquisition, including performance tuning and training to identify the most effective configurations. The final phase involved deploying the system on real hardware and refining the algorithms based on experimental results. In the future, this work may also connect with ongoing research at LIRMM on tactile sensing for humanoid manipulation, potentially forming the basis for the PhD continuation.

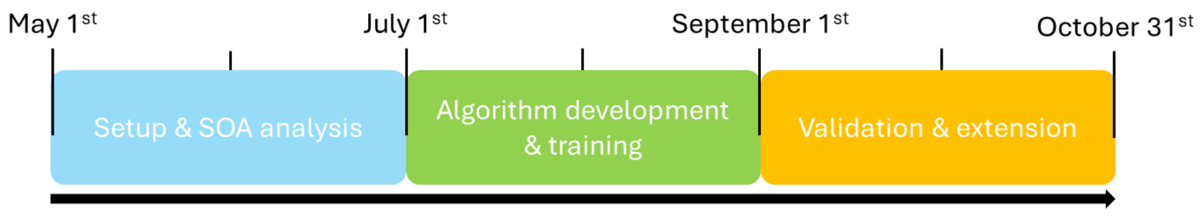


Figure 1: Internship workflow (6 Months).

1.2 Background and Motivations

The challenge of enabling a humanoid robot hand to learn dexterous object manipulation is a problem of high relevance for service robotics, teleoperation, and assistive technologies, but also one of the most difficult due to the complexity of contact-rich interactions, high-dimensional action spaces, and the need for generalization to unseen objects [**7**], [**8**], [**9**]. Artificial intelligence, particularly reinforcement learning and deep neural networks, offers promising avenues for enabling autonomous acquisition of such skills [**10**], [**11**]. However, adapting these methods to a humanoid robot capable of using self-body contacts and external forces to manipulate remains an open and demanding research problem.

In the case of Honda, this research aligns with the company's long-term vision of developing versatile humanoid robots for both industrial and service applications. Mastering dexterous manipulation could make factories more flexible, reducing the need for costly reconfiguration when changing production tasks, and ultimately supporting a shift toward more responsive, on-demand manufacturing systems. The first challenge given by Honda was manipulating multiple screws with a single hand while bringing one screw to a desired state (specific position and orientation). This task, illustrated in **Figure 2**, requires highly dexterous manipulation, not only through precise finger contacts but also by leveraging external forces (human-like interactions with the environment) to guide or separate screws before grasping, as the fingers alone may not have the fine control needed to directly achieve the target state.

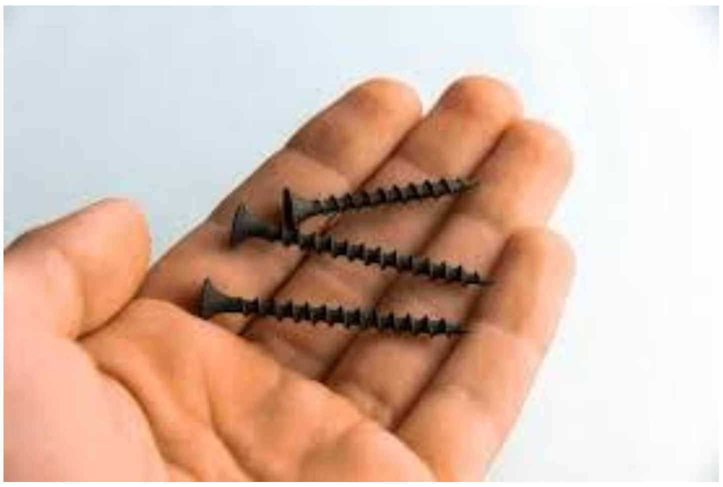


Figure 2: Example of a human hand performing in-hand screws manipulation.

1.3 Report Outline

This report presents the work conducted during the internship, focusing on Reinforcement Learning (RL) approaches for dexterous object manipulation with a humanoid robot hand. In the first part, the report reviews the state of the art in manipulation, covering both model-based and learning-based approaches, and identifies the remaining challenges. It then details the chosen simulation environment, agent architectures, and training procedures, highlighting key results and observed limitations, which motivate the transition toward continual learning. The second part addresses continual learning strategies, starting with a review of existing approaches, including evolving architectures and fixed architectures strategies. It then introduces a novel learning architecture, presenting the theoretical basis, weight representation methods, and experimental outcomes. Finally, the report discusses current conclusions, remaining limitations, and ongoing investigations into automatic continual learning methods using hypernetworks, outlining perspectives for future research.

2 Reinforcement Learning Experiments

2.1 State of the Art in Object Manipulation

Dexterous object manipulation has been a long-standing challenge in robotics research, requiring precise control of contact forces, dynamic coordination of multiple joints, and the ability to generalize to novel objects and tasks [**7**], [**8**], [**9**]. Two broad approaches dominate the literature: model-based and learning-based methods [**10**].

2.1.1 Model-Based Approaches

Model-based manipulation relies on accurate kinematic and dynamic models of the robot and the manipulated objects. Classical control techniques such as inverse kinematics/dynamics, model predictive control (MPC), and impedance/admittance control have been successfully applied to industrial arms and some anthropomorphic hands. Notable examples include:

1) Universal Robots' UR Series: These articulated robotic arms utilize inverse kinematics and impedance control to perform precise assembly tasks in unstructured environments. [**12**]
2) Baxter Robot: Baxter employs model-based control strategies, including MPC, to adapt to dynamic environments and perform complex manipulation tasks. [**13**]
3) KUKA's LBR iiwa: This lightweight robotic arm integrates impedance control with real-time force sensing to interact with humans and handle delicate objects. [**14**]

These methods benefit from predictable, explainable behavior and can achieve high precision when the model is accurate. However, they often require detailed object geometry, exact mass/inertia parameters, and reliable contact models, conditions that are rarely met in unstructured or dynamic environments. Moreover, modeling whole-body humanoid manipulation with multiple contacts (including self-body contacts) quickly becomes intractable due to high dimensionality and discontinuities in contact dynamics.

2.1.2 Learning-Based Approaches

Learning-based methods, especially those using reinforcement learning (RL) and deep neural networks, bypass the need for an explicit analytical model by directly learning policies from data or simulation. Model-free RL approaches (e.g., PPO [**15**], SAC [**16**], TD3 [**17**]) have shown promising results in complex in-hand manipulation tasks, as demonstrated by systems like OpenAI's Dactyl [**18**] and NVIDIA's Isaac Lab-based dexterous control [**19**]. Model-based RL blends learned dynamics models with planning (e.g., PETS [**20**], Dreamer [**21**]), improving sample efficiency at the cost of added modeling complexity. Additionally, imitation learning and demonstration-augmented RL have been used to bootstrap training with expert trajectories [**22**], [**23**], [**24**]. Despite their flexibility, these methods face challenges in sim-to-real transfer, training stability, and scaling to whole-body control involving arms, torso, and legs.

2.1.3 Remaining Challenges

Current research identifies several persistent challenges [**7**], [**10**], [**11**]:

- **Contact modeling:** Accurately representing multi-contact interactions, including self-body contacts, remains difficult both in simulation and reality.
- **High-dimensional control:** Humanoids with dexterous hands have dozens of actuated degrees of freedom, making exploration and policy learning slow and unstable.
- **Generalization:** Policies trained for specific objects or tasks often fail to transfer to new ones without retraining.
- **Sample efficiency:** Model-free RL typically requires millions of interactions, which is impractical for real-world deployment.
- **Sim-to-real gap:** Differences between simulated physics and real-world dynamics lead to degraded performance when deploying on hardware.

Addressing these limitations motivates the experimental work presented in this report, where simulation-based RL training is used as a foundation for investigating continual learning strategies to improve adaptability and long-term skill retention.

2.2 Selected Simulation Environment and Agents

In RL (Reinforcement Learning), an agent learns to act in an environment by trial and error, receiving rewards that guide it toward desirable behaviors. Over time, the agent improves its policy[1] such that the hand can perform increasingly complex manipulation tasks. The experimental setup for this work was designed to support scalable and reproducible training of dexterous manipulation policies for a humanoid robot hand. This section presents the simulation framework, environment configuration, and the reinforcement learning agents selected for the study.

2.2.1 Framework and Environment Setup

All simulations were conducted using the NVIDIA Isaac Lab framework based on Isaac Sim simulator [**25**], chosen for its ability to handle contact-rich, high-degree-of-freedom systems with GPU-accelerated physics. While the use of a whole-body humanoid robot has been considered to learn our dexterous manipulation using self-body contacts, we first limited our training environment to a multi-fingered hand. The model used is the Allegro Hand V4 [**26**] for which a model is provided by one of the Isaac Lab environment example, **Figure 3.a** shows the Allegro Hand V4 and **Figure 3.b** shows the corresponding model in Isaac Lab. We chose this hand to be as close as possible to the one Honda is currently developing. Then, as shown **Figure 3.c**, we extended it with a 3-DoF wrist to enable complex in-hand manipulation using external forces such as gravity and inertia. Also, the hand has 3D tactile sensors on each fingertip.

[1] A policy in reinforcement learning is a function that determines the agent's actions given its current observations of the environment.

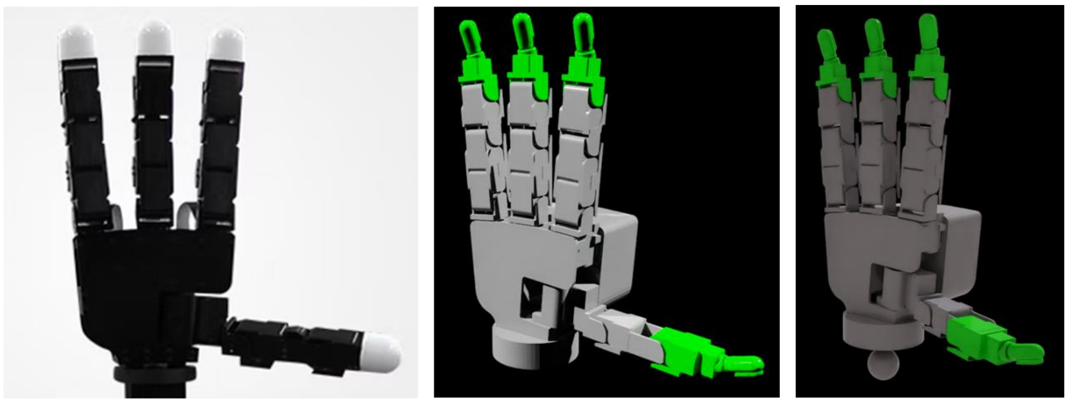

a) Allegro Hand V4 (real hardware)
b) Allegro Hand model in Isaac Sim
c) Extended model with 3-DoF wrist

Figure 3: Allegro Hand V4 visuals.

The default environment provided by NVIDIA is designed to train a robotic hand to move a cube to a desired orientation and position. As an initial step, we evaluated its performance to establish a baseline, given that this environment has already been demonstrated to be learnable. Building upon it, we adapted the setup to our use case, which requires manipulation of multiple objects within the hand. **Figure 4** illustrates our parallelized environment within Isaac Sim, implemented using the Isaac Lab framework. **Figure 5** presents a side-by-side comparison of the characteristics of the original Isaac Lab environment and our extended version. The table for our extension highlights the final configuration that produced the most promising results. Throughout the internship, we performed numerous training runs to identify appropriate reward functions, observation spaces, and action representations that ensured stable and efficient learning for our use case. To make the differences from the default environment more visible, we applied a color scheme in the table of our extension: green indicates newly introduced terms, red marks removed terms, and blue denotes modified terms.

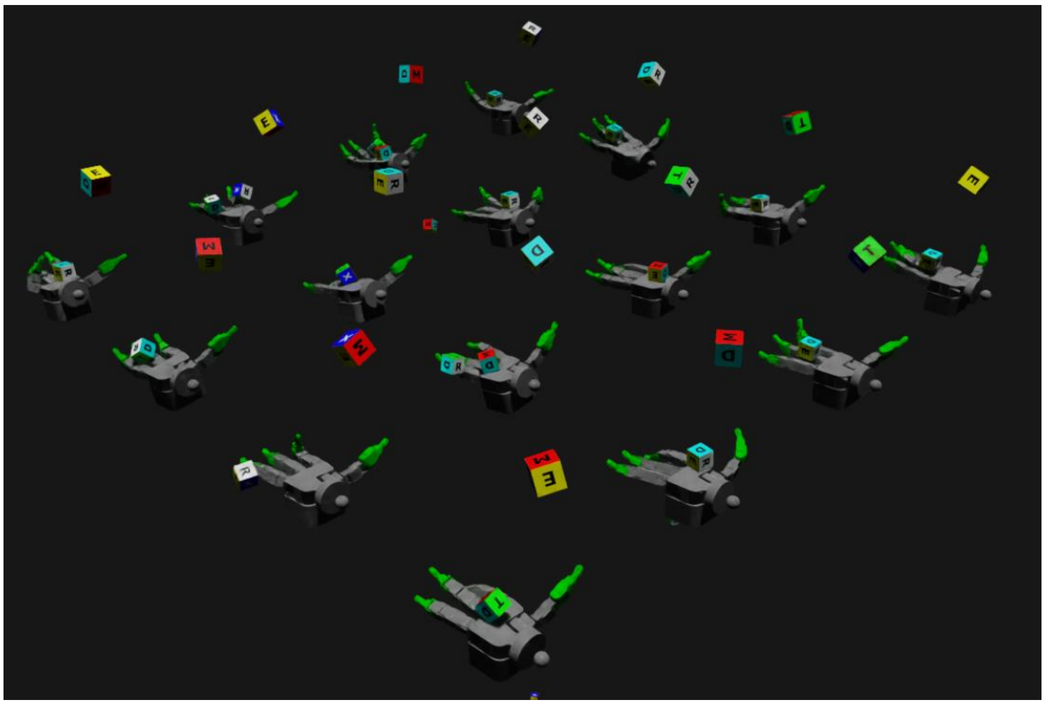
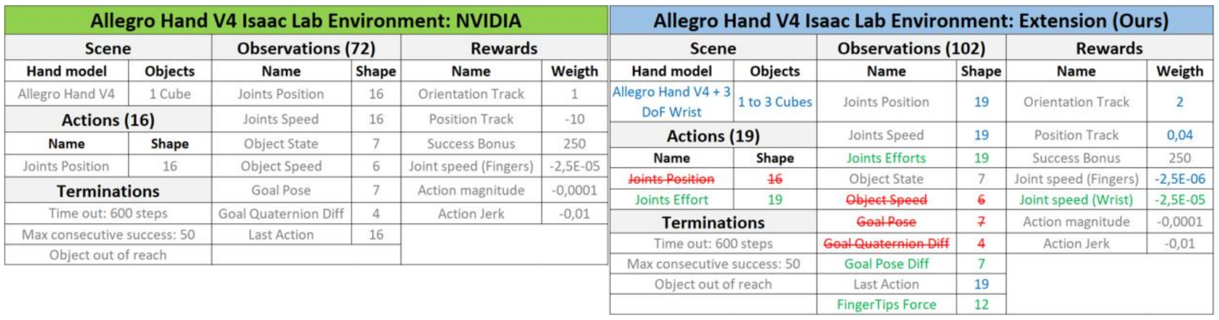


Figure 4: Allegro Hand V4 parallelized environment in Isaac Sim.



Allegro Hand V4 Isaac Lab Environment: NVIDIA					
Scene		Observations (72)		Rewards	
Hand model	Objects	Name	Shape	Name	Weigth
Allegro Hand V4	1 Cube	Joints Position	16	Orientation Track	1
Actions (16)		Joints Speed	16	Position Track	-10
Name	Shape	Object State	7	Success Bonus	250
Joints Position	16	Object Speed	6	Joint speed (Fingers)	-2,5E-05
Terminations		Goal Pose	7	Action magnitude	-0,0001
Time out: 600 steps		Goal Quaternion Diff	4	Action Jerk	-0,01
Max consecutive success: 50		Last Action	16		
Object out of reach					

Allegro Hand V4 Isaac Lab Environment: Extension (Ours)					
Scene		Observations (102)		Rewards	
Hand model	Objects	Name	Shape	Name	Weigth
Allegro Hand V4 + 3 DoF Wrist	1 to 3 Cubes	Joints Position	19	Orientation Track	2
Actions (19)		Joints Speed	19	Position Track	0,04
Name	Shape	Joints Efforts	19	Success Bonus	250
~~Joints Position~~	~~16~~	Object State	7	Joint speed (Fingers)	-2,5E-06
Joints Effort	19	~~Object Speed~~	~~6~~	Joint speed (Wrist)	-2,5E-05
Terminations		~~Goal Pose~~	~~7~~	Action magnitude	-0,0001
Time out: 600 steps		~~Goal Quaternion Diff~~	4	Action Jerk	-0,01
Max consecutive success: 50		Goal Pose Diff	7		
Object out of reach		Last Action	19		
		FingerTips Force	12		

Figure 5: Isaac Lab NVIDIA's example environment VS ours.

As detailed in **Figure 5**, training a reinforcement learning (RL) agent on an environment requires defining three key components: the observations, which serve as inputs to the policy; the rewards, which guide the agent toward the desired behavior; and the actions, which represent how the policy's outputs interact with the environment.

Observations: We followed NVIDIA's default setup by including joint positions and velocities, and then extended it by adding joint efforts and fingertip forces. This augmentation enables the agent to gather information about multiple objects held in the hand without explicitly providing their pose or orientation. The only object represented in the observations is the target object to be manipulated. This design both facilitates learning and yields a realistic model capable of handling an unrestricted number of additional objects within the same hand. Furthermore, to improve realism, we removed object velocity from the observations, since in real-world scenarios it is typically a noisy measurement, whereas object position and orientation can be estimated with sufficient accuracy through visual feedback. Moreover, the goal specification was reformulated from an absolute goal pose to a goal pose difference, such that the agent learns to drive an object in the desired direction rather than memorizing specific target positions. This formulation promotes generalizable directional control, enabling the policy to adapt to varying object placements and initial conditions while maintaining efficient and stable learning.

Rewards: The most critical aspect of training is reward shaping. We experimented with several configurations, introducing new terms and adjusting weights, but the configuration that yielded the best results, shown in **Figure 5**, remains largely similar to NVIDIA's default. The main change concerns position tracking: previously, the reward was formulated as a penalty (the farther the object is from the target position, the higher the penalty), whereas we switched to a positive reward formulation (the closer the object is to the target, the higher the reward). Interestingly, this change led to faster and more stable learning.

Figure 6 compares the corresponding learning curves: although the absolute reward values are not directly comparable, the difference in learning speed is evident. With the penalty formulation (left), training progresses much more slowly, whereas with the positive reward formulation (right), the agent learns significantly faster. One possible explanation is that positive rewards provide a clearer and more consistent learning signal, as the agent receives immediate reinforcement for moving toward the goal rather than being primarily penalized for errors. This can reduce negative gradient effects and encourage exploration in directions that improve performance, resulting in more efficient policy optimization.

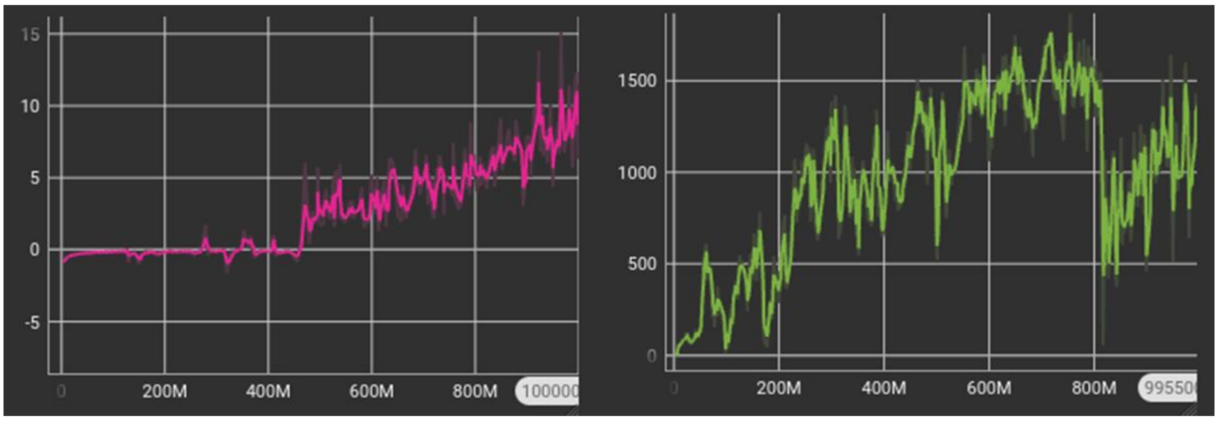


Figure 6: Comparison of learning curves with penalty-based (Left) vs. reward-based (Right) position tracking.

Actions: Actions were changed from position control to effort/torque control, allowing the agent to exploit contact dynamics and interact more naturally with multiple objects in the hand. This formulation enables more dexterous and human-like manipulation, as the policy can leverage forces to adjust grips, separate objects, and guide them toward the desired configuration, rather than relying solely on precise joint positions.

Finally, as shown in **Figure 4**, to accelerate learning and leverage NVIDIA's GPU-accelerated platform Isaac Sim, we trained multiple environment instances in parallel.

2.2.2 Agent Architectures

Configuring the environment is only one part of the process; equally important is selecting an appropriate training strategy. Among the reinforcement learning algorithms commonly applied to continuous control, Proximal Policy Optimization (PPO) [**15**] and Soft Actor-Critic (SAC) [**16**] are particularly prominent, as they combine robustness with strong performance in high-dimensional tasks.

Proximal Policy Optimization (PPO) is an on-policy[2] actor–critic[3] method that improves the policy while preventing large, destabilizing updates. It achieves this by maximizing a clipped surrogate objective, described below:

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

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Where:

- $\hat{E}_t$ is the empirical expectation over several data,
- θ are the parameters of the policy network being optimized,
- $r_t(\theta)$ is the probability ratio comparing how likely the new policy is to take action a_t in state s_t relative to the old policy,
- $\hat{A}_t$ is the estimated advantage function (detailed below),

[2] Learns from data collected by the same policy it is improving.
[3] Combines a policy network (actor) that selects actions with a value network (critic) that evaluates them.

- ϵ is a hyperparameter that defines the clipping range, limiting the size of policy updates to stabilize training.

A key distinction from typical reinforcement learning loss functions is the use of the advantage estimate $\hat{A}_t$, This term represents the difference between the discounted return, detailed in **Appendix 1**, obtained from the agent's behavior under the policy (the actor), and the baseline value estimate provided by a second neural network (the critic) which also learns from the collected data. So $\hat{A}_t$ measures how much better an action a_t was, compared to the expected reward. It serves as a weight that amplifies or downplays the importance of each sampled action when updating the policy, thereby guiding learning toward behaviors that perform better than expected.

Then, the use of $min(.)$ and $clip(.)$ functions constrains how much the probability ratio between the new and old policies can deviate. This regularization stabilizes training by preventing overly aggressive updates, thereby reducing the risk of catastrophic performance drops. Such stability is especially important in contact-rich tasks, where even small changes in the policy can lead to drastically different dynamics.

PPO is widely appreciated for its robustness, relatively simple hyperparameter tuning, and its ability to produce smooth and physically plausible behaviors in complex control settings such as humanoid locomotion. The full theoretical details are beyond the scope of this report; interested readers can refer to the original paper [**15**] or the explanatory video [**27**].

Soft Actor-Critic (SAC) is an off-policy[4] algorithm based on the maximum entropy reinforcement learning framework [**16**]. In addition to maximizing task reward, SAC encourages policies with higher entropy[5], promoting exploration and preventing premature convergence to suboptimal deterministic strategies. SAC's off-policy nature allows it to reuse past experiences efficiently, making it more sample-efficient than PPO. However, its performance can be sensitive to hyperparameter tuning, and it may require careful reward shaping in complex manipulation tasks.

In this work, PPO was selected as the primary training algorithm due to its stability in large action spaces, relatively straightforward hyperparameter tuning, and widespread adoption in robotic control research. SAC was considered as an alternative, but after some experiments, PPO's predictable convergence behavior made it a better fit for iterative experimentation and later integration with continual learning methods.

As with the environment, several PPO architectures and hyperparameter settings were explored. Once a stable configuration was identified (shown in **Table 1**) we kept it fixed and instead concentrated on refining the environment through modifications to rewards, observations, and actions.

The policy architecture was selected through empirical trials, balancing the need for sufficient capacity to learn the task with the requirement to avoid excessive network size and training time. To this end, we employ two separate Multilayer Perceptrons (MLPs) [**28**] to implement the policy

[4] Learns about one policy while using data from another.
[5] Entropy is a measure of randomness or uncertainty.

(actor) and the value function (critic). A general overview of how are structured MLPs can be found in **Appendix 2**.

Name	Value	Description
Architecture (actor & critic)	MLP [512, 256, 128] Activation: ELU Optimizer: Adam	The configuration of the neural networks (number of layers and neurons) that encode the policy and value functions.
Batch size	16 384	The number of experience samples used simultaneously to compute a gradient update.
Nb steps	24	The number of environment interactions collected before performing policy updates.
Nb epochs	5	The number of times each batch of experience collected is reused to optimize the policy and value networks.
Log STD	1.0	The logarithm of the policy's action distribution standard deviation, controlling exploration magnitude in continuous action spaces.
Learning rate	5e-4	The scalar step size applied to parameter updates during optimization, controlling the speed and stability of learning.

Table 1: PPO architecture and hyperparameters chosen.

Both networks share the same hidden-layer configuration of three layers with 512, 256, and 128 neurons, respectively (see **Table 1**). Given that our environment provides 102 observations and requires 19 actions (see **Figure 5**), the input and output layers of the policy network are dimensioned accordingly. In an MLP, every neuron is fully connected to all neurons in the adjacent layers (see **Appendix 2**). Training the policy therefore corresponds to adjusting the weights and biases of these connections so that, for a given set of observations, the network outputs the desired actions. This configuration results in 218 496 trainable weights and 915 biases, for a total of 219 411 parameters in the policy network. Similarly, the value network (critic) shares the same input and hidden layers architecture, but has a single output neuron, corresponding to the estimated state value used in the advantage calculation. Then this network comprises 216 211 trainable weights and 897 biases, totaling 217 108 parameters.

In total, the agent must learn 436 519 parameters across both networks. The procedure for training these networks using PPO from the Stable Baselines3 library [**29**] is described in the following section.

2.3 Training Procedures and Key Results

2.3.1 Learning strategy

MLP training can be significantly accelerated using Graphics Processing Units (GPUs) [**30**], as they are optimized for performing the large number of parallel matrix multiplications and vector operations required in neural network computations. This parallelism allows faster forward and backward passes during training compared to traditional CPUs. To leverage this advantage, we conducted our trainings on two separate computers equipped with dedicated GPU cards. The specifications of these machines are summarized in **Table 2**.

	Computer 1	Computer 2
Model	Dell Inc. Precision 7560	Dell Inc. Precision 7550
Graphic Card	NVIDIA RTX A4000	NVIDIA Quadro RTX 5000 Mobile / Max-Q
VRAM (GPU Dedicated) (MB)	8192	16384
NVIDIA Driver | CUDA Version	550.163.01 | 12.4	570.169 | 12.8
Processor	11th Gen Intel Core i9 - 2.60GHz x 16	Intel Xeon - 2.40GHz x 16
System RAM (MB)	32000	32000
OS	Unbuntu 22.04	Ubuntu 22.04

Table 2: System specifications of the training platforms.

For both computers, the trainings were performed inside a Docker[6] container on Ubuntu 22.04 ensuring that the working environment remains identical regardless of the machine used to run it. This guarantees reproducibility across experiments and simplifies the transfer of the setup to multiple computers. With this approach, we could reliably scale our experiments, training up to 5000 parallel environments on the first computer and up to 8 000 on the second. The training process is illustrated in **Figure 7**.

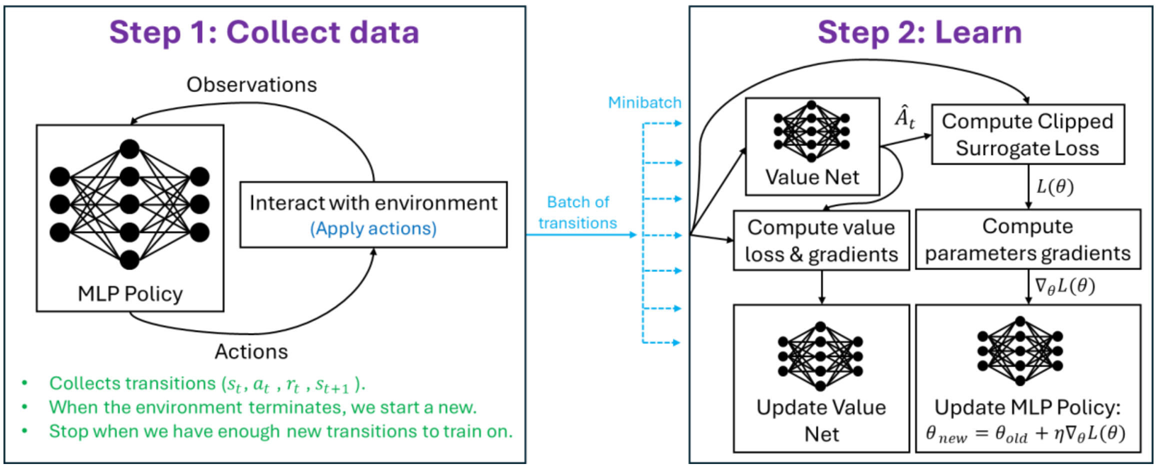

Step 1: Collect data
Observations
MLP Policy
Interact with environment
(Apply actions)
Actions
Collects transitions (s_t, a_t, r_t, s_{t+1}).
When the environment terminates, we start a new.
Stop when we have enough new transitions to train on.
Batch of transitions
Minibatch
Step 2: Learn
Value Net
$\hat{A}_t$
Compute Clipped Surrogate Loss
$L(\theta)$
Compute value loss & gradients
Compute parameters gradients
$\nabla_\theta L(\theta)$
Update Value Net
Update MLP Policy:
$\theta_{new} = \theta_{old} + \eta \nabla_\theta L(\theta)$

Figure 7: Schematic illustration for the training process of a MLP policy with PPO.

Step 1: The first step consists of collecting a batch of data, represented as transitions (s_t, a_t, r_t, s_{t+1}), corresponding respectively to the observations at time t, the action chosen by the policy based on those observations, the reward received from that action, and the resulting next observations at time $t+1$. The agent collects data for a specified number of steps (see *'Nb steps'* in **Table 1**), and the total number of transitions collected per step equals the number of parallel environments. For instance, on the second computer, using 24 steps with 8 000 parallel environments, the agent collects $24 \times 8\,000 = 192\,000$ transitions before starting a training update, corresponding to step 2 in **Figure 7**.

Step 2: During training, the agent does not update the policy using the entire collected batch at once. Instead, the collected data is divided into minibatches of a fixed size (a hyperparameter, see *'Batch size'* in **Table 1**). The policy is updated sequentially on each minibatch, and this process is repeated for multiple epochs (see *'Nb epochs'* in **Table 1**), meaning the same collected batch is used several times to improve learning stability and sample efficiency. In our case, we use 5 epochs. This approach ensures that updates remain on-policy while allowing multiple gradient

[6] Docker is a platform that packages software and its dependencies into isolated, reproducible environments.

steps over the same set of collected transitions. For each minibatch, the value network provides the state-value estimates used to calculate the advantages $\hat{A}_t$ and the Clipped Surrogate Loss $L(\theta)$ defined in **Section 2.2.2** is computed. The gradients of this loss with respect to the policy parameters θ are then calculated using backpropagation[7]. These gradients are used by an adaptive optimizer (e.g., Adam [**31**] in our case, see *'Architecture'* in **Table 1**) to perform gradient ascent on θ, updating the policy parameters with a specified learning rate η (see *'Learning rate'* in **Table 1**). The learning rate controls the size of each update step: a large rate can destabilize training, while a too small rate slows convergence. As explained in **Section 2.2.2**, maximizing the Clipped Surrogate Loss $L(\theta)$ guides the agent toward behaviors that yield higher rewards, while following the gradient ascent on $L(\theta)$ encourages the policy to prefer actions leading to these rewards. This highlights why reward shaping is critical in RL, as it provides the primary feedback for the agent to improve its behavior.

Then, For the value network (critic), the process is slightly different. The critic is trained using supervised learning[8] to predict the expected return of a state. Its target values come from the data collected during training. By learning to estimate these values accurately, the critic provides a stable baseline that reduces variance in policy updates, improving overall learning stability. It is optimized in parallel with the policy, using the same minibatch and optimizer configuration.

Once the networks are updated, the agent collects a new batch of transitions using the updated policy (Step 1 in **Figure 7)**, and the cycle repeats until cumulative rewards are maximized.

2.3.2 Results & Comments

Training with this strategy typically required between 1 and 2 days to achieve the first signs of successful behaviors, while longer runs, up to 10 days, were sometimes preferred to obtain more consistent and meaningful results. **Table 3** presents the learning curves from some of our best training runs. The accompanying snapshots illustrate the state of the environment when agents either successfully reached their goals or failed by remaining blocked in suboptimal positions. In these images, the orientation target is represented by the floating cube above the hand. Some training sessions were carried out in multiple stages by reusing the trained weights from previous runs. In such cases, several learning curves are displayed for the same training session, each numbered to indicate the corresponding training stage. To facilitate comparison, modifications to the training setup relative to the previous stage are highlighted in purple.

Training 1: The first training used NVIDIA's default Allegro Hand model and configuration (see **Figure 5**) to establish a baseline for acceptable learning times and performance with cube orientation. The training was performed in three stages, simply because the initial limits were insufficient, as rewards kept improving after each stage. The snapshots show that the hand successfully learned to grasp and rotate the cube into the target orientation.

[7] **Backpropagation:** Standard algorithm for computing the gradients of a neural network's loss function with respect to its weights.
[8] **Supervised Learning:** Supervised learning refers to training a model using labeled input-output pairs, where the model learns to predict the output from the input. In contrast, reinforcement learning does not provide explicit labels; instead, an agent learns by interacting with an environment and receiving scalar reward signals that indicate the quality of its actions.

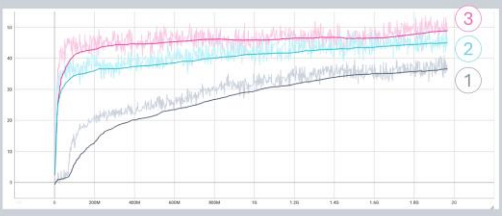
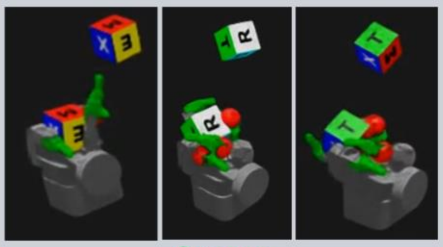
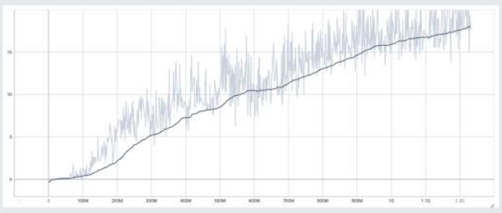
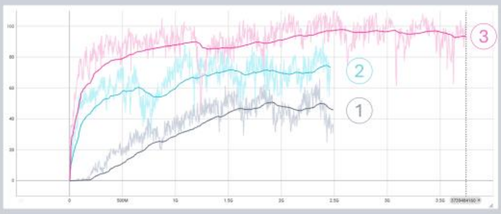
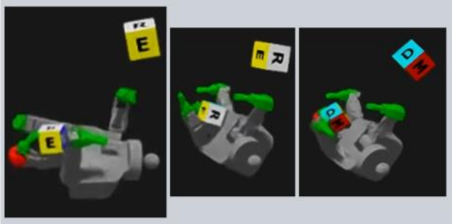
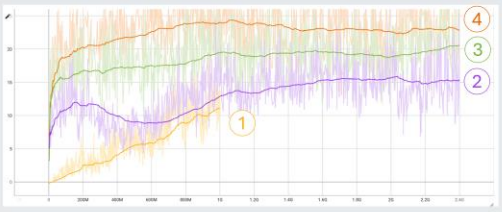
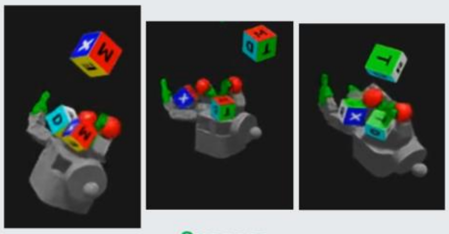
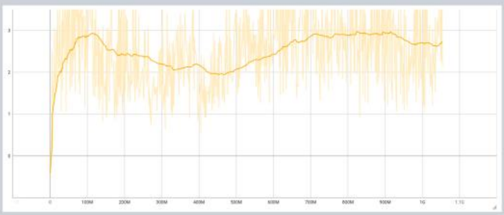
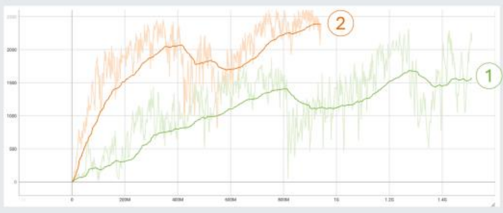
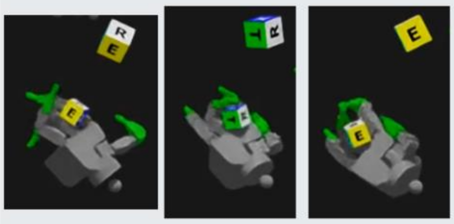
Training Setup	Rewards	Results
1 **Goal:** Orientation **Environment:** Allegro hand V4 1 Cube NVIDIA Configuration	 **5.89G timesteps | 8.4 Days**	 **Success**
2 **Goal:** Orientation **Environment:** Allegro hand V4 + Wrist (90° limit) 1 Cube NVIDIA Configuration	 **1.23G timesteps | 1.7 Days**	 **Success**
3 **Goal:** Orientation **Environment:** Allegro hand V4 + Wrist (45° limit) 1 Cube Our Configuration	 **8.69G timesteps | 10.3 Days**	 **Success**
4 **Goal:** Orientation **Environment:** Allegro hand V4 + Wrist 2 Cubes Our Configuration	 **8.20G timesteps | 7.1 Days**	 **Success**
5 **Goal:** Orientation **Environment:** Allegro hand V4 + Wrist 3 Cubes Our Configuration	 **1.05G timesteps | 1 Day**	 **Fail**
6 **Goal:** Position + Orientation **Environment:** Allegro hand V4 + Wrist 1 Cube Our Configuration	 **2.45G timesteps | 1.9 Days (Still training)**	 **Partial**

Table 3: RL training results and environment outcomes.

Training 2: We then extended the default NVIDIA model with a wrist to investigate whether the agent could exploit gravity and inertia during manipulation, as humans often do in multi-object tasks. The 3-DoF wrist was limited to ±90° rotations on all axes. Learning was straightforward, but the agent did not use gravity as expected; instead, it relied on grasping the cube and rotating it with the wrist. While this strategy achieved reward improvements, it was unsatisfactory for our use case: it would not generalize to multi-object tasks and is constrained by the wrist's limited range of motion. Therefore, we stopped this training and adjusted the configuration.

Training 3: In this setup, we applied our own configuration (defined in **Figure 5**) and reduced the wrist rotation limit to ±45°, making the motion more realistic for integration into a humanoid forearm. This resulted in the desired behavior: instead of tightly gripping and rotating with the wrist, the agent learned to use gravity and inertia to roll the cube into the target orientation, then stabilized it with its fingers.

Training 4: After success with one cube, we extended the task to two cubes. As explained in **Section 2.2.1**, the agent only received feedback about one target cube, while the other cube's state had to be inferred through torque and fingertip sensors. This training produced very promising results: the agent first rolled the cubes to position the target cube in its palm, then used its fingers to orient it while simultaneously preventing the other cube from falling.

Training 5: We then investigated manipulation with three cubes, though their size made success unlikely even for a human. The reward curve quickly plateaued, and the snapshots show that the agent learned to grasp all three cubes to avoid early termination due to falling objects. However, rotating one cube into the desired orientation was too difficult, since all fingers were occupied with grasping. The agent attempted to use the wrist to compensate, but as in Training 2, this strategy was too limited to succeed.

Training 6: Finally, we extended the goal to include both position and orientation, since the ultimate task requires delivering an object in the correct pose (e.g., for screw positioning). Training is ongoing, but initial results are promising. The snapshots show that the hand maintains the cube in the target position, which is randomized on a plane above the hand level. The most effective strategy observed so far involves rolling the cube with gravity and inertia to align its orientation, then lifting and stabilizing it to meet the positional goal.

During our experiments, several observations and limitations became apparent:

- **Finger behaviors:** Fingers often adopted random poses and struggled with precise manipulations, highlighting the difficulty of learning fine-grained control using the current setup. Imitation learning could encourage the policy to adopt more natural and consistent behaviors, but this approach introduces additional challenges, such as generating suitable datasets and mapping human motions to the agent's action space.
- **MLP architecture constraints:** The selected Multilayer Perceptron (MLP) is fixed once trained, which imposes structural limitations. Modifying the observation or action spaces, adding new objects, or increasing the network's complexity (e.g., adding layers or neurons) requires careful redesign and retraining, often with extensive hyperparameter tuning. These constraints make it difficult to scale the policy to more

complex tasks or adjust it for future customization, independently of the policy's ability to retain previously learned skills.

Another central challenge observed is catastrophic forgetting: when attempting to reuse a policy trained on a single cube to accelerate learning with two or three cubes, the agent often exhibited incorrect behaviors and lost previously learned skills. This highlights the classic trade-off between plasticity (learning new tasks) and stability (retaining old knowledge). In the context of object manipulation, this is particularly critical: policies trained on one object type (e.g., cubes) frequently fail when confronted with different shapes or physical properties (e.g., screws).

One potential approach to mitigate this issue is to employ a very large network and train it simultaneously on all tasks. However, this quickly leads to slow learning, longer execution times, and scalability problems, especially given the virtually infinite possibilities in object manipulation scenarios. Another strategy is to combine learning-based methods for high-level decision-making with model-based controllers for low-level execution [**32**]. While powerful, such analytical approaches can be complex to implement, sensitive to model inaccuracies, and computationally expensive, particularly in contact-rich environments.

Alternatively, continual learning and meta-learning techniques aim to develop policies that can adapt over time and transfer skills between tasks without retraining from scratch. In our specific context, given that Honda operates in controlled factory environments with known object models, a model-based approach may be the most practical solution. Nevertheless, this internship explored continual learning strategies as a pathway toward more adaptive and versatile manipulation skills, with the goal of eventually enabling policies that can generalize across tasks while reusing prior knowledge and avoid catastrophic forgetting.

3. Continual Learning Approaches

3.1 Overview of Continual Learning

Continual learning (CL) refers to the ability of an artificial agent to acquire new skills or adapt to new tasks over time without catastrophically forgetting previously learned knowledge. This challenge, known as catastrophic forgetting, occurs when standard gradient-based training overwrites parameters that were critical for earlier tasks. In robotic manipulation, the ability to retain and integrate past experiences is crucial for enabling robots to expand their skillset, adapt to new objects or environments, and operate effectively over extended periods without having to be retrained from scratch.

Broadly, continual learning strategies fall into two main paradigms: those that expand their architecture as new tasks arrive, and those that maintain a fixed architecture while protecting or modulating existing parameters [**33**].

3.1.1 Growing/Evolving Architectures

Growing or evolving architectures address continual learning by progressively increasing the model's capacity as new tasks are introduced. This can be achieved by adding neurons, layers, or even entire subnetworks dedicated to the new skills. A notable example is Progressive Neural Networks (Rusu et al., 2016) [**34**], in which each new task is assigned its own network "column" connected laterally to previous ones to enable transfer of useful features without overwriting them. **Figure 8** illustrates the architectural scheme from the original paper.

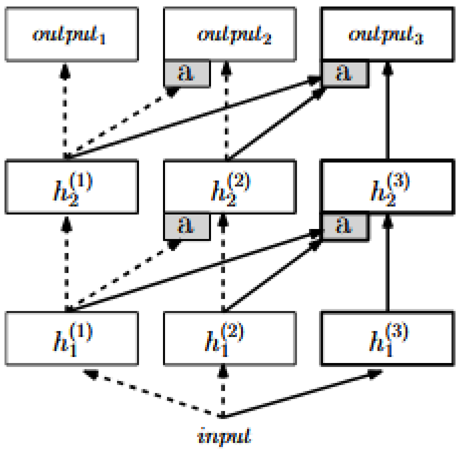


Figure 8 (From original Paper [34]**): Depiction of a three-column progressive network. The first two columns on the left (dashed arrows) were trained on task 1 and 2 respectively. A third column is added for the final task having access to all previously learned features.**

Some approaches expand the network only when a drop in performance signals insufficient capacity, thereby balancing learning flexibility with computational cost. For instance, the Dynamically Expandable Network (DEN) [**35**] model selectively retrains and expands its architecture only when necessary, avoiding unnecessary complexity and computational overhead. Similarly, the Self-Evolved Dynamic Expansion Model (SEDEM) [**36**] evaluates the diversity among subnetworks to control model growth, ensuring efficient continual learning without excessive resource consumption. These strategies enable models to adapt to new tasks while maintaining performance on previously learned tasks, addressing the challenges of catastrophic forgetting and limited model capacity. However, they come at the expense of unbounded memory growth and often require explicit knowledge of which task is being performed at inference time, which can limit their applicability in autonomous, open-ended settings [**33**].

3.1.2 Fixed Architectures

Fixed-architecture approaches tackle the problem from a different angle by keeping the network size constant while introducing mechanisms to prevent interference between tasks. One family of methods uses regularization to discourage changes to parameters deemed important for previous tasks, as in Elastic Weight Consolidation (EWC) [**37**], where importance is estimated using the Fisher information matrix[9]. Another related method is Synaptic Intelligence (SI) [**38**], which tracks the importance of each parameter online during training and penalizes updates to weights critical for previous tasks, offering a more flexible and computationally efficient alternative to EWC.

Another family relies on parameter isolation, where different subsets of neurons or weights are dedicated to different tasks to prevent interference. Binary or sparse masks are often applied to

[9] **Fisher information matrix:** A measure of the sensitivity of a model's output to changes in its parameters, often used to estimate the importance of each parameter for preserving previously learned knowledge in continual learning.

selectively activate the relevant parameters for each task, effectively “switching on” only the portions of the network needed. Examples include PackNet (Mallya et al., 2018) [**39**], which iteratively prunes and reassigns network weights for new tasks, and Piggyback (Mallya et al., 2018) [**40**], which learns task-specific binary masks over a fixed backbone network. These approaches allow the network to retain previously learned behaviors without interference, while reusing shared parameters when possible.

A more recent direction involves dynamic modulation, in which an auxiliary model, often called a hypernetwork, generates task-specific weights or modifications for the main network on demand. Instead of directly storing separate parameters for each task, the hypernetwork outputs either full or partial weight sets conditioned on a task embedding. This allows the main network to adapt its behavior dynamically without overwriting its shared parameters, effectively mitigating catastrophic forgetting while keeping memory usage bounded. Such methods have been explored in works like Ha et al. (2016) [**41**], who introduced hypernetworks as a general mechanism to generate the weights of a target network from a smaller network, demonstrating flexibility across different tasks and architectures, and von Oswald et al. (2020) [**42**], who applied hypernetworks for continual learning by producing task-specific weights to mitigate catastrophic forgetting. **Figure 9** illustrates the working principle of a hypernetwork: the task embedding is fed into the hypernetwork, which outputs task-specific weights that modulate the main network for that task.

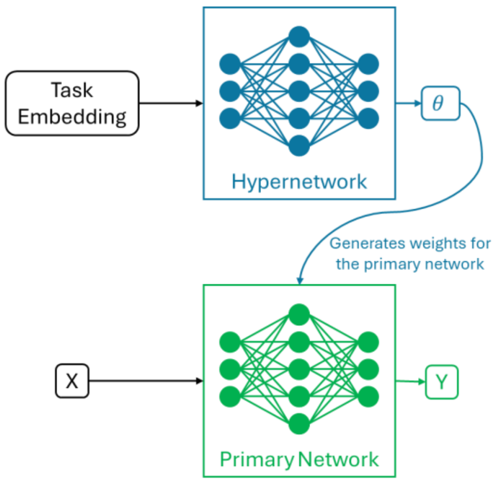


Figure 9: Conceptual scheme of a hypernetwork.

3.1.3 Open Challenges

Despite these advances, continual learning in robotics still faces several unresolved issues [**33**], [**43**]. Real-world deployments rarely provide explicit signals about when a task changes, meaning that agents must autonomously detect and adapt to new situations without supervision. The stability–plasticity dilemma also persists: a robot must remain flexible enough to rapidly acquire new skills while preserving those already mastered. As noted through our experiments, reusing a policy trained for manipulating a single cube often led to catastrophic forgetting when extended to more complex tasks (e.g., two or three cubes), illustrating the difficulty of scaling to richer scenarios. Methods discussed earlier, such as Progressive Neural Networks or Hypernetworks, offer partial solutions but face limitations: they may grow unboundedly, rely on clearly defined task boundaries, or struggle with the high-dimensional continuous control required for dexterous manipulation. Efficiently transferring knowledge between tasks without interference thus remains an open challenge.

3.2 Proposed Learning Architecture

To tackle the limitations identified in **Section 3.1**, we began developing a centralized architecture designed to integrate multiple models while enabling smooth knowledge sharing. Unlike the fixed MLP used in our baseline, this framework is not bound to a single observation or action space, nor to a frozen topology. Instead, it stores different policies within a unified representation where parameters are expressed as continuous functions. This additional functional dimension allows policies to interpolate and exchange parameters across models regardless of their size or complexity, whether small, lightweight networks or larger, deeper architectures, providing a mechanism for both specialization and transfer. By extending the representation space in this way, the architecture aims to mitigate catastrophic forgetting while preserving the flexibility to incorporate future tasks without retraining from scratch.

To formalize our approach, let us first recall some definitions about Neural Networks:

- $l \in \mathbb{N}$: index of a layer, where $l = 0$ is the input layer and L is the output layer.
- $n^l \in \mathbb{N}$: number of neurons of layer l; n^0 being the number of inputs or observations.
- $X \in \mathbb{R}^{n_0}$: input vector containing the observations.
- $Y \in \mathbb{R}^{n_L}$: output vector containing the actions in the case of RL.
- $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: weights matrix between layer l.
- $b^{(l)} \in \mathbb{R}^{n_l}$: bias vector of layer l.
- $a(.)$: activation function (e.g., ReLU, tanh).
- $h^{(l)} \in \mathbb{R}^{n_l}$: hidden state vector at layer l.
- $\theta^{(l)}$: parameters $W^{(l)}$ and $b^{(l)}$ at a layer l.

3.2.1 Theoretical Foundations and Intuition

The main limitation of classic MLP networks in a continual learning setting is that they rely on a fixed architecture with static parameters. At the neuron level, the computation can be written as (see **Figure 10.a** for a visual representation):

$$\mathrm{h}(W, b) = a(WX + b) = a\left(\sum_{0}^{\mathrm{n}} w_i x_i + b\right)$$

With $n = \dim(X)$, $W \in \mathbb{R}^{\mathrm{n}}$, $b \in \mathbb{R}$ and $\mathrm{h}(W, b) \in \mathbb{R}$ here as we are considering only one neuron.

At the network level, the forward pass through an L-layer feedforward neural network is given by (see **Figure 10.b** for a visual representation):

$$h^{(l)}(\theta^{(l)}, h^{(l-1)}) = a\big(W^{(l)} h^{(l-1)} + b^{(l)}\big), \ \ l = 1, \dots, L$$

With $h^{(0)} = X$ the inputs and $Y = h^{(L)}$ the outputs.

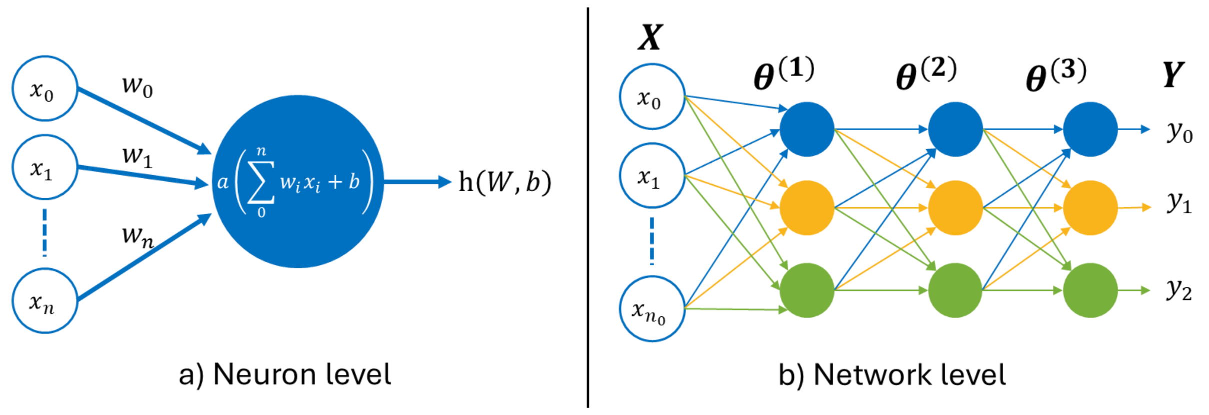


Figure 10: Visual representation of a classic Neural Network forward pass.

Here, the parameters $\theta^{(l)}$ are fixed once the training phase is completed. This rigidity limits the model's ability to adapt to new tasks without retraining and makes it prone to catastrophic forgetting when sequentially exposed to multiple tasks.

Our first intuition to extend this capability is to represent the parameters not as static values but as functions. Instead of assigning a constant weights or bias to each neuron, we let these parameters vary smoothly with respect to an auxiliary variable z, which may encode task identity, context, or interpolation along a policy space. At the neuron level, the new formulation becomes:

$$\mathrm{h}(F_w, Z_w, f_b, z_b) = a((F_w \circledast Z_w)X + f_b(z_b)) = a\left(\sum_0^{\mathrm{n}} f_{wi}(z_{wi})x_i + f_b(z_b)\right)$$

Where:

- $\circledast$ is defined as the element-wise application operator so that:
  If we have $F = \big(f_1(.), f_2(.), \dots, f_n(.)\big)$ a vector (or matrix) of functions and $X = (x_1, x_2, \dots, x_n)$ a vector (or matrix) of values of the same shape as F, such that each $f_i(x_i)$ is well-defined. Then: $F \circledast X = \big(f_1(x_1), f_2(x_2), \dots, f_n(x_n)\big)$.
- $n = \dim(X)$: the number of observations.
- $F_w \in \mathbb{R}^{\mathrm{n}}$: the vector of functions f_{wi} for weights.
- $Z_w \in \mathbb{R}^{\mathrm{n}}$: the vector of auxiliary variables z_{wi} for F_w.
- f_b: the function for the bias of the neuron.
- $z_b \in \mathbb{R}$: the auxiliary variable for f_b.

and at the network level:

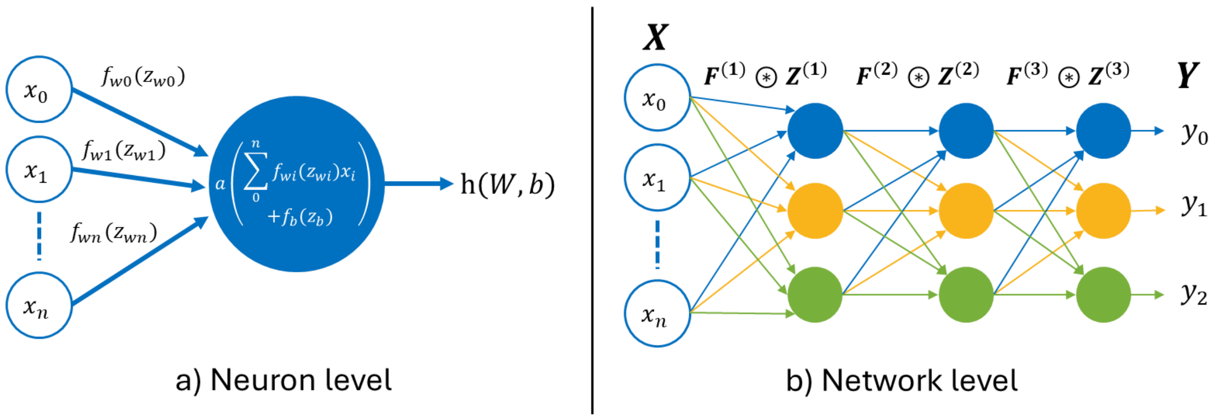
$$h^{(l)} = a\left(\left({F_w}^{(l)} \circledast {Z_w}^{(l)}\right) h^{(l-1)} + \left({F_b}^{(l)} \circledast {Z_b}^{(l)}\right)\right), \; l = 1, \dots, L$$

$$X = h^0;\; Y = h^L$$

With :

- ${F_w}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: the matrix of functions f_{wi} for the weights at layer l.
- ${Z_w}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: the matrix of auxiliary variables for ${F_w}^{(l)}$.
- ${F_b}^{(l)} \in \mathbb{R}^{n_l}$: the vector of functions f_{bi} for the bias at layer l.
- ${Z_b}^{(l)} \in \mathbb{R}^{n_l}$: the vector of auxiliary variables for ${F_b}^{(l)}$.

Figure 11 illustrates this extended formulation.



x_0
x_1
x_n
$f_{w0}(z_{w0})$
$f_{w1}(z_{w1})$
$f_{wn}(z_{wn})$
$a\left(\sum_0^n f_{wi}(z_{wi})x_i + f_b(z_b)\right)$
$\mathrm{h}(W, b)$
a) Neuron level
$\boldsymbol{X}$
$\boldsymbol{F}^{(1)} \circledast \boldsymbol{Z}^{(1)}$
$\boldsymbol{F}^{(2)} \circledast \boldsymbol{Z}^{(2)}$
$\boldsymbol{F}^{(3)} \circledast \boldsymbol{Z}^{(3)}$
$\boldsymbol{Y}$
y_0
y_1
y_2
b) Network level

Figure 11: Visual representation of our extended Neural Network forward pass.

This reformulation effectively introduces a third dimension into the neural network: instead of mapping only from inputs to outputs, the model also operates over an additional functional space using auxiliary variables z for each parameter. In this extended space, policies can adapt, interpolate, and transfer knowledge across tasks without discarding previously acquired information.

- If z encodes a task index, the same network can express multiple task-specific policies without retraining.
- If z interpolates between tasks, the model can smoothly transition between different policies.
- If z encodes context, the network dynamically adapts its behavior to changing environments.

In this way, the network gains flexibility for continual learning, knowledge transfer across tasks, and resilience to catastrophic forgetting.

3.2.2 Parameters Representation as Functions

We chose to represent the parameters (weights w_i and biases b_i in the case of a classical neural network) as continuous functions to allow smooth navigation along the additional functional dimension introduced in **Section 3.2.1**. Instead of storing parameters as fixed scalar values, the system now stores centers (encoded by the auxiliary variable z) associated with specific magnitudes (the parameter values to be preserved). A function $f(.)$ is then required to continuously interpolate between these stored points.

Several families of function approximators can be considered for this purpose. The most straightforward choice are parametric functions, such as low-degree polynomials [**44**] or splines [**45**], which offer simple closed-form behavior but often lack flexibility and generalization capacity when a large variety of parameter transitions must be represented. Another option is to use piecewise functions [**46**] which provides direct control over stored points but tends to introduce discontinuities or overshooting when extrapolating outside the sampled regions.

A more advanced family includes kernel-based methods, where the contribution of each stored point is weighted by a kernel function depending on its distance to the query point. This provides locality and smoothness, while also allowing the representation to adapt as new centers are added. Within this family, Radial Basis Functions (RBFs) [**47**] are particularly attractive because they can approximate arbitrary smooth functions given a sufficient number of centers, and their behavior can be tuned directly via the choice of kernel shape and bandwidth.

Other approaches, such as Fourier series [**48**] expansions or neural implicit functions [**49**] (small MLPs that themselves learn to approximate the parameter mapping), could also be considered. However, these either require global adjustments that affect the entire functional space (as in Fourier or polynomial expansions) or introduce an additional optimization layer that complicates training (as in neural implicit functions).

For our use case, we required a representation that is:

1) **Local**, to ensure that adding new points minimally perturbs existing knowledge.
2) **Continuous and smooth**, to enable stable interpolation between parameters.
3) **Efficient**, since the system must be queried at every forward pass of the network.

These constraints naturally motivated us to start with kernel-based formulations, and in particular RBF. Which representation for a function $f(z)$ can be written as:

$$f(z) = \sum_{j=1}^{M} \alpha_j k(\|z - c_j\|)$$

Where M is the number of centers, c_j are the centers, α_j are the magnitude values, and $k(.)$ is the kernel function. We tested several kernels, such as the Gaussian and the Multiquadric, but ultimately retained the Gaussian kernel because of its smooth decay, its ability to represent parameter transitions without abrupt changes, and its natural tendency to vanish in regions far from any stored center. This latter property is particularly desirable in our setting: we want areas of the functional space that have not been sampled or learned yet to default gracefully to zero, rather than introducing oscillations or extrapolated values. The Gaussian kernel is defined as:

$$k(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right)$$

Where σ controls the width of the kernel.

While RBFs provided encouraging results, they also introduced several limitations for our use case. The first drawback was the appearance of artifacts when neighboring centers were too close or when parameter values between centers differed significantly. In such cases, undesired oscillations could occur (see **Figure 12.a**). Adjusting kernel widths to mitigate these effects often resulted in a trade-off, making it difficult to preserve a simple Gaussian-like behavior without undesirable influence on unexplored areas between two centers.

A second drawback concerns computational efficiency. Classical RBF implementations require evaluating contributions from all centers when computing $f(z)$. As the number of stored points grows, this quickly becomes impractical. Moreover, most existing libraries provide little control over restricting evaluations to the most relevant centers, which is essential for scalability in a continual learning setting.

To overcome these limitations, we implemented a custom distance-aware linear interpolation method with Gaussian decay that borrows intuition from lookup-based interpolation [**50**], while retaining the smoothness of kernel methods. Specifically, instead of considering all stored centers, our method relies only on the two closest centers to the query point z. First, their parameter values are interpolated linearly, as in a standard lookup-based scheme. Then, this interpolation is modulated by a Gaussian decay factor that smooths the transition and reduces sharp discontinuities. The resulting formulation is:

$$f(z) = \left(\alpha_{j1}(1-\lambda) + \alpha_{j2}\lambda\right) \cdot \exp\left(-\frac{d(z)^2}{2\sigma^2}\right)$$

With:

- $j1, j2$: the indices of the two nearest centers to z.
- $d(z) = \min\left(|z - c_{j1}|, |z - c_{j2}|\right)$: the minimum distance from z to either center.
- $\lambda = \frac{\|z - c_{j1}\|}{\|c_{j2} - c_{j1}\|}$: the normalized linear interpolation factor.

This lightweight scheme retains the efficiency and interpretability of linear interpolation, but the Gaussian decay smoothly attenuates the influence of distant points, both in interpolation and extrapolation regimes. It combines the simplicity of lookup-based piecewise linear methods with the smoothness control characteristic of kernel-based approaches, where the decay factor directly regulates the transition sharpness. As illustrated in **Figure 12.b**, our method avoids the unwanted oscillations observed with standard RBF interpolation, while producing smooth and accurate transitions between neighboring centers.

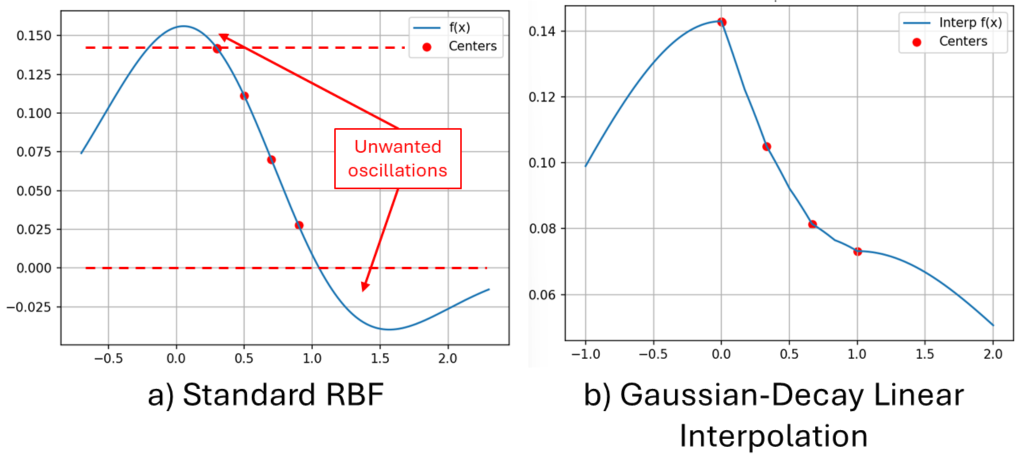


Figure 12: Comparison of parameter interpolation methods: Standard RBF vs. Gaussian-Decay Linear Interpolation.

In practice, this ensures that adding a new center affects only its local neighborhood and does not unintentionally alter far-away values, a crucial property when the function is used to represent neural network parameters across tasks. Moreover, in estimation experiments with up to 1 000 points, standard RBF interpolation slowed from 1 000 Hz to 200 Hz, whereas our scheme maintained stable performance around 1 300 Hz.

3.2.3 Implementation

Two main learning strategies were investigated during the internship. The first aimed to learn the parameter functions directly online, replacing the classic methods and optimizers used for fixed weights. This approach is illustrated in **Figure 13**. However, despite the custom Gaussian decay being efficient even with thousands of points, this method slowed down training and could not match the speed of GPU-accelerated fixed-weight training. The second strategy leverages the efficiency of tensor-based learning for fast training, and then wraps the learned results into functional representations for knowledge sharing or other continual learning strategies. In this way, our architecture can be seen as a wrapper around standard training. This approach is illustrated in **Figure 14**.

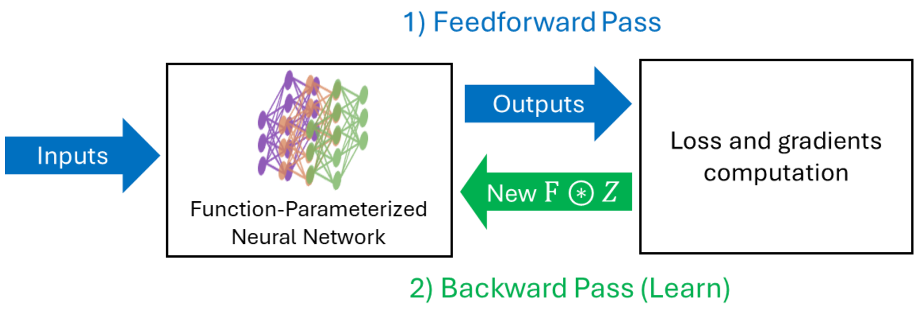


Figure 13: Learning Strategy: Direct Functional Learning.

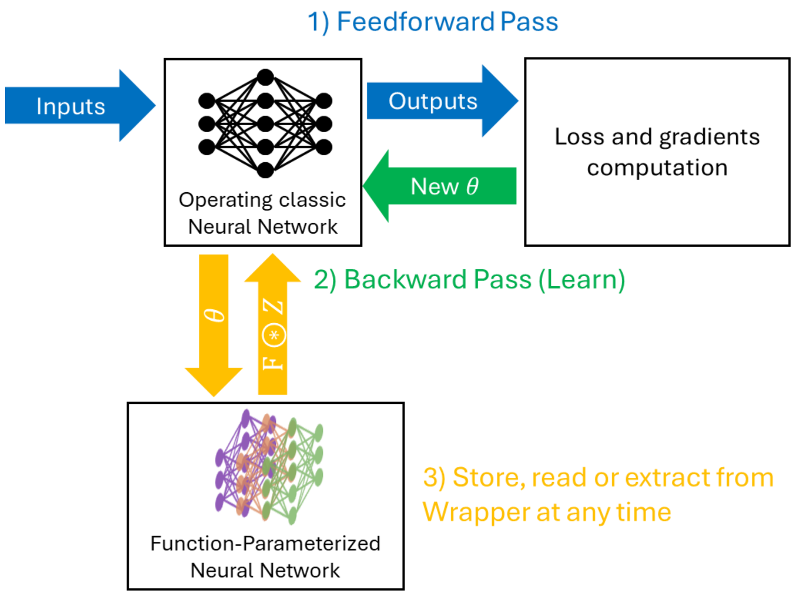

1) Feedforward Pass
Inputs
Operating classic Neural Network
Outputs
New θ
Loss and gradients computation
2) Backward Pass (Learn)
θ
$F \circledast Z$
Function-Parameterized Neural Network
3) Store, read or extract from Wrapper at any time

Figure 14: Learning strategy: Wrapper approach.

Using the wrapper-based method offers several advantages. Any existing pre-trained network can be wrapped to benefit from the additional functional dimension introduced by our architecture. The wrapper allows parameters to be manipulated online, stored, or extracted at any stage, providing flexibility to enhance continual learning strategies. It remains highly adaptable, as the wrapper creates a unified space in which different models can interact, share knowledge, or be compressed. For instance, higher-level strategies such as hypernetworks can learn how and when to leverage the wrapped representations. Furthermore, both offline and online optimization can be applied on the wrapper, enabling centralized management of multiple learned models and facilitating efficient knowledge transfer across tasks.

3.4 Experimental Results & conclusions

3.4.1 Theory validation

The first validation aimed to verify that our wrapper can correctly store multiple models and allow reliable extraction for runtime use. In this experiment, we created three models with different architectures: a medium-sized model with three hidden layers of 64 neurons each ([64, 64, 64]), a smaller model with [12, 12], and a larger model with [128, 64, 86, 24]. These models were stored in the wrapper using the simple strategy illustrated in **Figure 15**, where each model's parameters are represented as squares for simplicity, and overlapping areas indicate parameters sharing the same continuous dimension in the wrapper.

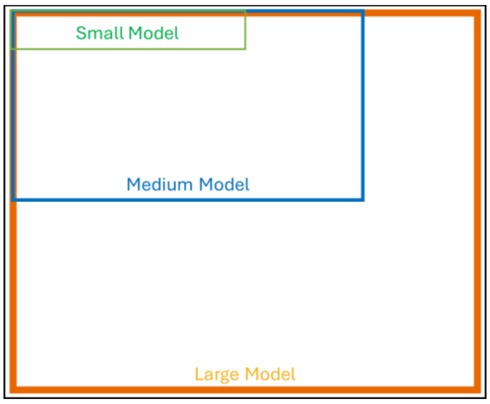


Figure 15: Illustration of a possible Wrapper storage strategy for multiple models.

After storing the models, we extracted each one to verify that the recovered parameters matched the original agents. The medium model was associated with the encoding auxiliary variable $z = 0.0$, the small one with $z = 1.0$, and the largest with $z = 3.0$. As shown in **Figure 16.a**, the wrapper weight function f_{w0} contains three continuously connected weights. For the medium model, some functions contain only two connected weights (**Figure 16.b**), since the smaller model cannot share all its parameters with the medium model. Similarly, for the largest model, certain functions store only a single weight (**Figure 16.c**) for the same reason.

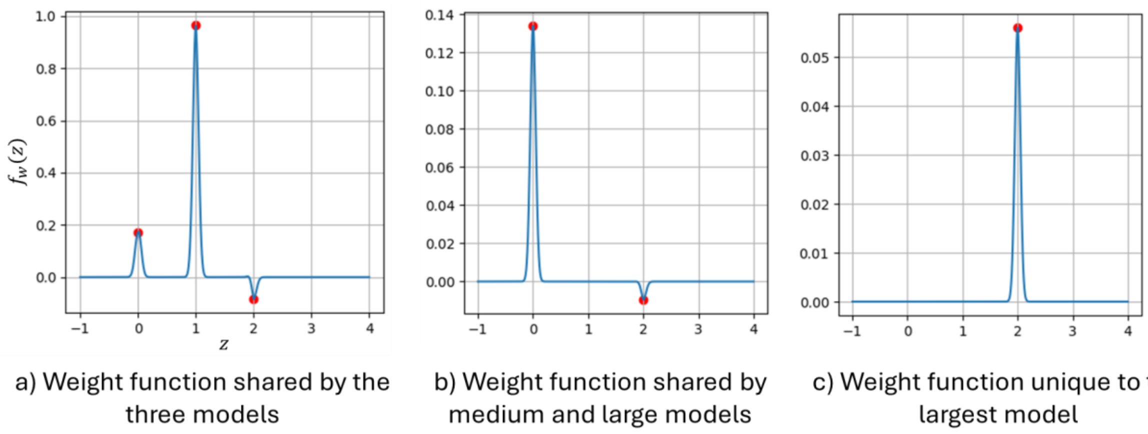


Figure 16: Visualization of weight functions in the Wrapper after storing models of different architectures.

Although the models are embedded according to the default strategy in **Figure 15**, custom storage or extraction behaviors can be defined. The key requirement is that the user or the employed strategy (e.g., a hypernetwork) retrieves the stored parameters correctly.

These results successfully validate the wrapper's ability to merge models with different architectures while introducing a third functional dimension for knowledge sharing and flexibility of the strategy employed.

3.4.2 Knowledge sharing

The second experiment aimed to evaluate how our wrapper can facilitate knowledge transfer across tasks. Specifically, we tested whether a model could leverage previously learned knowledge as a starting point for training on a new, related task.

We considered a simple case of the CartPole environment from Gymnasium [**51**] using a PPO agent from the Stable Baselines3 library [**29**] with a minimal MLP architecture: one hidden layer with a single neuron. (For details on the environment, observations, rewards, and termination conditions, see the Gymnasium documentation [**52**]). The mass of the pole was varied across four experiments: $0.1\ Kg$ (default), $0.8\ Kg$, $1.2\ Kg$, and $2.0\ Kg$.

Several agent were trained with the auxiliary variable z encoding the pole mass ($z = 0.1$ for $0.1\ Kg$, $z = 0.8$ for $0.8\ Kg$, etc.). **Figure 17** shows that we could correctly learn and store distinct agents for each mass that maximize rewards with values largely above 195 which is the threshold to consider the agent successful.

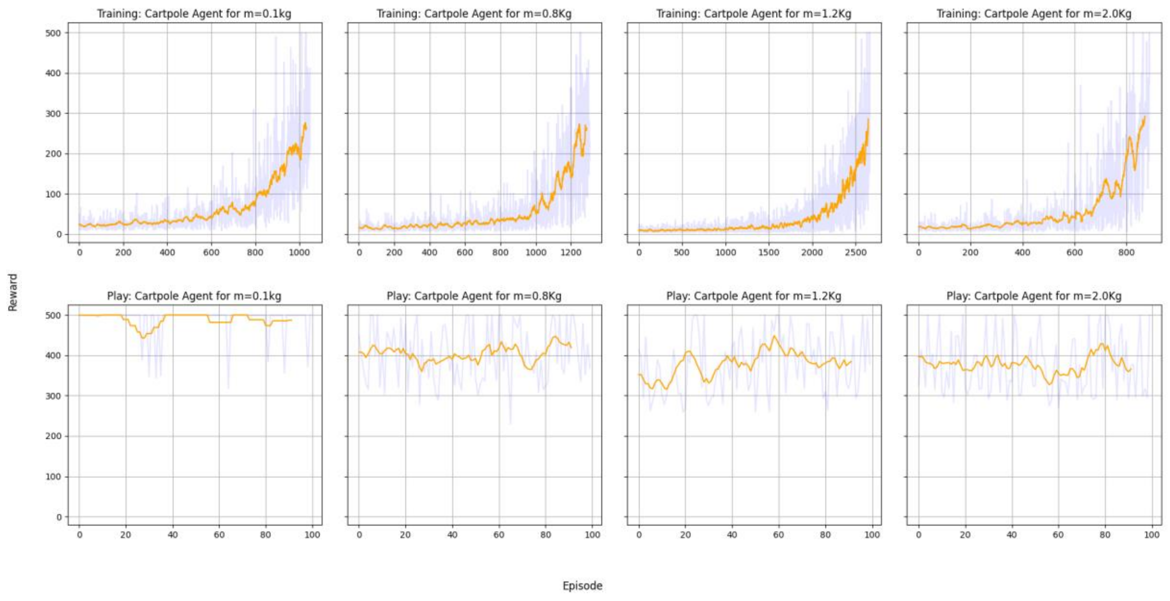


Figure 17: Learning and playing multiple agents for different Pole masses using the Wrapper (reward threshold = 195).

Initially, each model had to learn from scratch. But using our wrapper flexibity, we then trained the agents sequentially: after each training, the resulting model was wrapped, so the next training could benefit from previously learned weights. **Figure 18** and **Table 4** show that this approach accelerates learning for all masses except 2.0 kg. For this mass, which differs significantly from the others, starting from default parameters initialization proved more effective than transferring prior knowledge. These results highlight that successful continual learning requires a higher-level strategy to decide when and how to leverage previous knowledge, e.g., a hypernetwork that selectively uses prior experiences for each task, and our wrapper facilitates the implementation of such strategies.

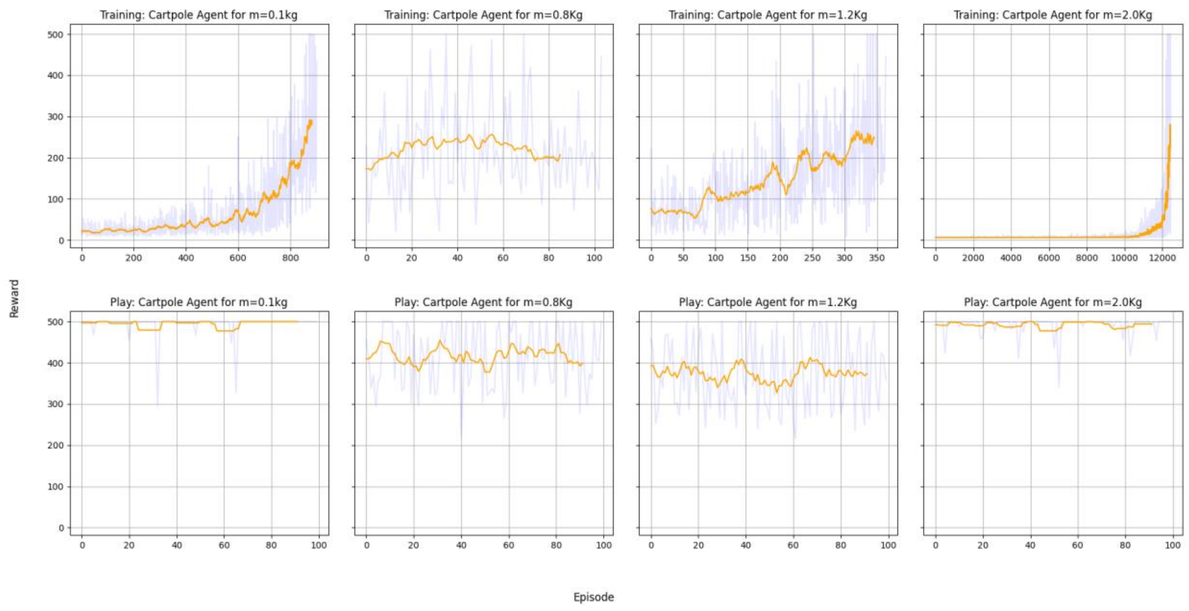


Figure 18: Sequential training with Wrapper-based knowledge transfer across Pole masses.

(Mean over 5 trainings for each)	From Scratch	With Knowledge transfer	Performance
Training time (s) m = 0.8 Kg	115.96	32.60	**-72%**
Training time (s) m = 1.2 Kg	150.48	61.00	**-59%**
Training time (s) m = 2.0 Kg	76.77	221.64	**+189%**
Overall Training time (s)	343.21	315.25	**-8%**

Table 4: Impact of sequential training with Wrapper-based knowledge transfer on learning speed across different Pole masses.

3.4.3 Model compression

The last experiment consisted of investigating if we can compress model knowledge using our wrapper rather than using classical networks. An agent with the minimal architecture of one hidden layer and one neuron was first trained on the default mass ($0.1\ Kg$), and its performance was then evaluated on the unseen masses for 100 episodes for each case. **Figure 19** shows the training results and performance across different masses.

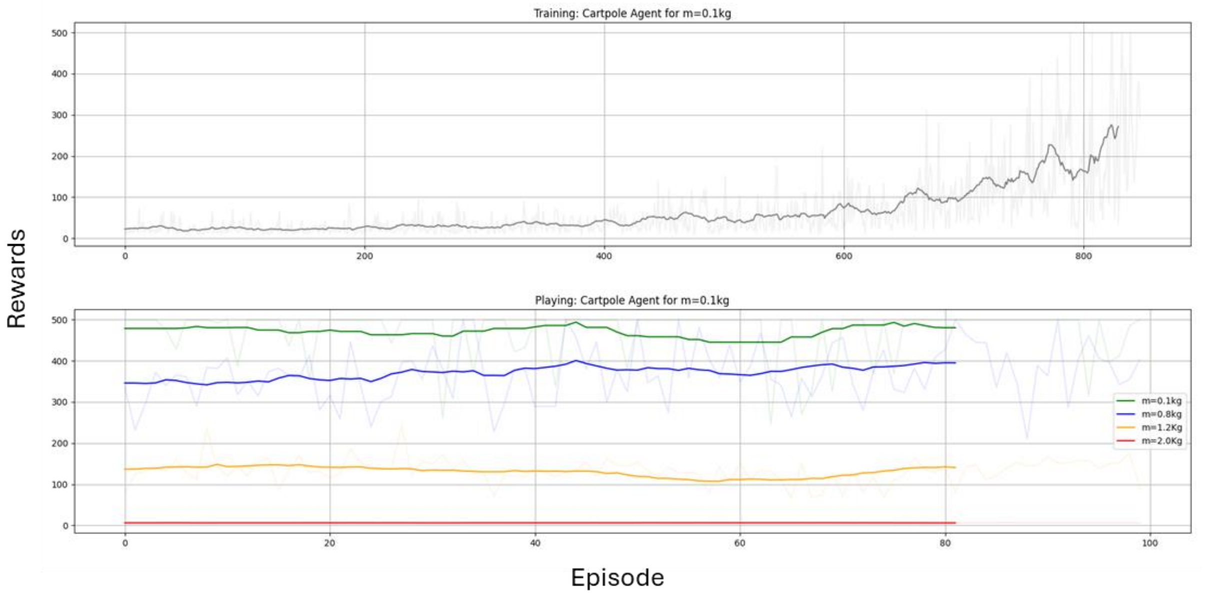


Figure 19: Performance of a PPO agent trained on a single Pole mass and tested on different masses.

As expected, such a simple model architecture trained on a specific mass fails to generalize to other masses, with performance degrading as the mass difference increases. This issue can be addressed using our wrapper with manual task embeddings, as described in **Section 3.4.2**. Since each minimal model has 16 parameters, storing four separate models would require at most $16 \times 4 = 64$ parameters in the wrapper, assuming no weight sharing between models.

We then trained an agent while progressively increasing its architecture until it achieved sufficient rewards across all four mass values. In this setting, during the training the mass was randomized at the end of each episode among the four possible values. The results are shown in **Figure 20**.

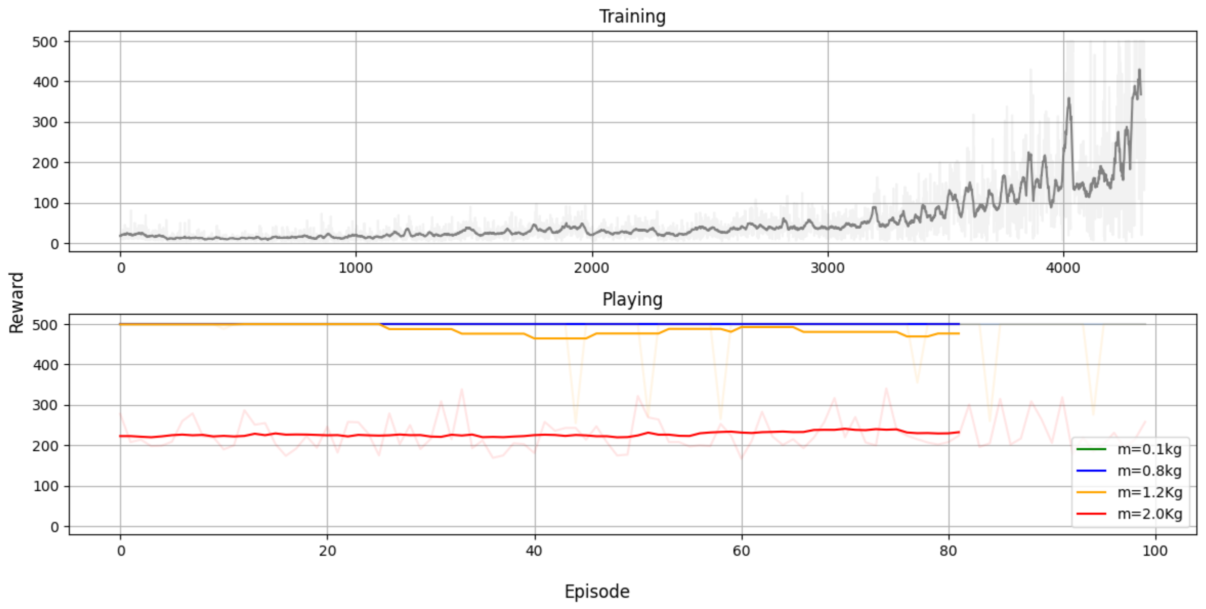


Figure 20: Classical Network training across varying Pole masses.

The observed rewards are comparable to those obtained with our wrapper using task embeddings for mass across all cases except 2 kg. For this mass, the rewards exceed the acceptable threshold, but the performance quality is still lower than when using our wrapper. The minimal architecture found for the classical network consisted of two hidden layers with two neurons each, totaling 103 parameters, about 60% more than the combined parameters of the task-specific models in our wrapper. For larger networks, this difference can become even more pronounced, potentially leading to substantial increases in both training and inference time.

These results demonstrate that our wrapper can successfully compress knowledge across multiple tasks while maintaining, or even improving, the quality of learned policies. By storing separate task-specific models within a unified representation, we can achieve comparable or better performance than training classical networks individually, while reducing the total number of parameters.

These results are encouraging but should be interpreted with care. In the classical network, the agent directly observes the mass and must learn how to incorporate this information into its policy. By contrast, in the wrapper case, the mass is explicitly provided as a task embedding, which likely accelerates the learning of each individual model. In real-world scenarios, such task embeddings are often non-trivial or even unavailable.

Finally, while our approach does not solve all continual learning challenges, it already demonstrates promising results even with simple strategies. The wrapper provides a solid foundation for future improvements by enabling models of different architectures to coexist within a unified representation, supporting both specialization and transfer. Its flexibility allows parameters to be stored, extracted, and manipulated across tasks, which opens the door to efficient reuse of knowledge, faster adaptation, and reduced memory overhead compared to retraining independent networks. At the same time, challenges such as managing the growth of stored representations, determining when and how to share parameters between tasks, and ensuring scalability to more complex environments remain open. Addressing these issues will likely require higher-level mechanisms, such as hypernetworks or meta-learning strategies, to guide the use of shared knowledge. Nonetheless, the results presented here position the wrapper as a promising foundation for building more advanced continual learning frameworks that balance adaptability, efficiency, and long-term knowledge retention.

4 Conclusions & Future work

This work explored the challenges of applying reinforcement learning to dexterous object manipulation using the Allegro hand. By leveraging scalable training infrastructures (Docker-based environments across multiple GPUs) we were able to train policies with thousands of parallel environments, achieving meaningful results within only a few days. The experiments demonstrated the potential of RL for learning non-trivial behaviors but also highlighted significant limitations of our baseline MLP policies, including poor generalization, rigid architecture constraints, and susceptibility to catastrophic forgetting when extending tasks beyond their original scope.

A review of the state of the art in continual learning confirmed that these issues are not specific to our setup but represent core challenges in the field: balancing plasticity and stability, efficiently transferring knowledge across tasks, and scaling methods to high-dimensional control domains. Existing solutions, such as growing architectures, regularization-based methods, parameter isolation, and hypernetworks, each provide partial answers but still struggle in realistic robotic scenarios where task boundaries are blurred and the variety of possible tasks is effectively unbounded.

To move beyond these constraints, we introduced the foundations of a centralized architecture capable of storing diverse models and facilitating seamless knowledge sharing by representing networks parameters as continuous functions. This design aims to combine adaptability with retention, paving the way toward policies that can incrementally accumulate skills instead of relearning them from scratch. While preliminary, this direction suggests a promising pathway for addressing lifelong learning in robotics.

Ultimately, building robots that can continuously learn and adapt in dynamic environments will require bridging the gap between current algorithmic advances and the complex realities of contact-rich manipulation. The contributions of this work provide both practical insights from large-scale RL experiments and a conceptual framework for architectures that may better support continual learning in the future.

Future work will follow two complementary directions. On the one hand, we will continue exploring classical reinforcement learning approaches combined with model-based controllers, targeting functional solutions for specific use cases such as screw manipulation in the Honda factory context. On the other hand, we will further investigate novel continual learning architectures to enable scalable, adaptive, and reusable skill acquisition across tasks.

This internship provided me with the opportunity to deepen my expertise in reinforcement learning, large-scale training infrastructures, and the challenges of continual learning. Building upon this work, I will continue the research as a PhD student starting in November, with the goal of advancing towards robotic systems that can learn continuously and adapt effectively to real-world industrial environments.

5. Acknowledgements

I would like to express my sincere gratitude to my supervisor, M. Kheddar ABDERRAHMANE, for his guidance and support throughout this internship, as well as to the Honda Research Institute in Tokyo for offering me the opportunity to work on such a stimulating and impactful project, and for supporting the continuation of this work through my upcoming PhD. I am also grateful to the academic staff of ENSTA Bretagne, and in particular those from the Robotics specialization, whose teaching and advice provided me with strong foundations and practical knowledge that I was able to apply directly in a professional research environment.

6. Appendix

Appendix 1: Discounted Rewards

In reinforcement learning, the discounted return from a timestep t is defined as:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Where:

- r_{t+k} is the reward received k steps after time t,
- $\gamma \in [0,1]$ is the discount factor, which reduces the contribution of future rewards compared to immediate ones.

The discount factor γ serves two purposes:

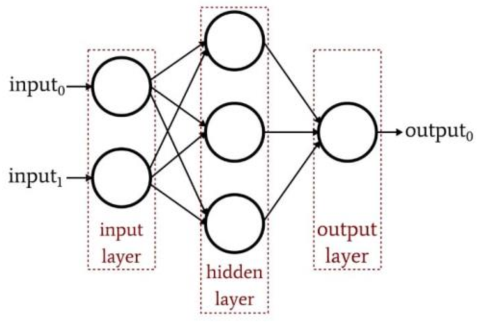
1. It encodes a preference for short-term rewards over distant ones (when $\gamma < 1$).
2. It ensures the infinite sum of rewards remains finite, stabilizing learning.

In practice, for finite episodes of length T, the return is computed as:

$$R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$$

Appendix 2: Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) is a type of feedforward artificial neural network commonly used to approximate complex functions in reinforcement learning and other machine learning tasks. It consists of an input layer, one or more hidden layers, and an output layer, with neurons in each layer fully connected to neurons in the adjacent layers.



Mathematically, the forward pass of an MLP can be described as:

$$h^{(1)} = a\left(W^{(1)}x + b^{(1)}\right)$$

$$h^{(l)} = a\left(W^{(l)}h^{(l-1)} + b^{(l)}\right), l = 2, \ldots, L-1$$

$$y = a(W^{(L)}h^{(L-1)} + b^{(L)})$$

Where:

- x is the input vector,
- $h^{(l)}$ is the activation vector of layer l,
- $W^{(l)}, b^{(l)}$ are the weight matrix and bias vector for layer l,
- $a(.)$ is the activation function (commonly ReLU or tanh) for hidden layers,
- L is the total number of layers including the output layer.

The training of an MLP consists in adjusting the weights $W^{(l)}$ and biases $b^{(l)}$ to minimize a loss function that measures the discrepancy between the network's predictions y and the desired outputs. In reinforcement learning, the outputs typically parameterize a policy distribution or a value function.

MLPs are flexible function approximators and can represent highly nonlinear mappings, making them suitable for approximating policies and value functions in complex environments.

7. References

[1] "LIRMM – Laboratoire Informatique Robotique et Microélectronique de Montpellier." [Online]. Available: https://www.lirmm.fr/

[2] "Honda Research Institute Japan Co., Ltd. - The official website of Honda Research Institute Japan Co., Ltd.," Honda Research Institute Japan Co., Ltd. [Online]. Available: https://www.jp.honda-ri.com/en/

[3] "Université de Montpellier," https://www.umontpellier.fr. [Online]. Available: https://www.umontpellier.fr

[4] "Centre national de la recherche scientifique (CNRS)." [Online]. Available: https://www.cnrs.fr/fr

[5] "IDH Team: Interactive Digital Humans – LIRMM." [Online]. Available: https://www.lirmm.fr/teams-en/idh-en/

[6] "What Is Reinforcement Learning?" [Online]. Available: https://www.mathworks.com/discovery/reinforcement-learning.html

[7] "The Developments and Challenges towards Dexterous and Embodied Robotic Manipulation: A Survey." [Online]. Available: https://arxiv.org/html/2507.11840v1

[8] A. Billard and D. Kragic, "Trends and challenges in robot manipulation," *Science*, vol. 364, no. 6446, p. eaat8414, Jun. 2019, doi: 10.1126/science.aat8414.

[9] A. G. Billard, "In good hands: A case for improving robotic dexterity," *Science*, vol. 386, no. 6727, p. eadu2950, Dec. 2024, doi: 10.1126/science.adu2950.

[10] C. Yu and P. Wang, "Dexterous Manipulation for Multi-Fingered Robotic Hands With Reinforcement Learning: A Review," *Front. Neurorobotics*, vol. 16, Apr. 2022, doi: 10.3389/fnbot.2022.861825.

[11] "A Careful Examination of Large Behavior Models for Multitask Dexterous Manipulation." [Online]. Available: https://toyotaresearchinstitute.github.io/lbm1/

[12] "Série UR." [Online]. Available: https://www.universal-robots.com/fr/

[13] “Le robot Baxter de Rodney Brooks pour la recherche et l’éducation.” [Online]. Available: https://www.generationrobots.com/fr/401514-baxter-robot-research-prototype.html?srsltid=AfmBOoosZZiI2z_ZesXh92_tpUyrafGo_GLN_nLsrSl1K0BwSUArrwnY
[14] “KUKA LBR iisy 15 R930,” Unchained Robotics. [Online]. Available: https://unchainedrobotics.deen//produkte/roboter/cobot/kuka-lbr-iisy-15-r930
[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Aug. 28, 2017, *arXiv*: arXiv:1707.06347. doi: 10.48550/arXiv.1707.06347.
[16] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” Aug. 08, 2018, *arXiv*: arXiv:1801.01290. doi: 10.48550/arXiv.1801.01290.
[17] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” Oct. 22, 2018, *arXiv*: arXiv:1802.09477. doi: 10.48550/arXiv.1802.09477.
[18] A. I. Advice, “Unveiling Dactyl: OpenAI’s Leap Forward in Robotic Learning and Automation,” Medium. [Online]. Available: https://medium.com/@AIadvice/unveiling-dactyl-openais-leap-forward-in-robotic-learning-and-automation-95e37664cd41
[19] “Available Environments — Isaac Lab Documentation.” [Online]. Available: https://isaac-sim.github.io/IsaacLab/main/source/overview/environments.html
[20] K. Chua, R. Calandra, R. McAllister, and S. Levine, “Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models,” Nov. 02, 2018, *arXiv*: arXiv:1805.12114. doi: 10.48550/arXiv.1805.12114.
[21] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, “Dream to Control: Learning Behaviors by Latent Imagination,” Mar. 17, 2020, *arXiv*: arXiv:1912.01603. doi: 10.48550/arXiv.1912.01603.
[22] H. Hu, S. Mirchandani, and D. Sadigh, “Imitation Bootstrapped Reinforcement Learning,” May 20, 2024, *arXiv*: arXiv:2311.02198. doi: 10.48550/arXiv.2311.02198.
[23] H. Zhao, X. Yu, D. M. Bossens, I. W. Tsang, and Q. Gu, “Beyond-Expert Performance with Limited Demonstrations: Efficient Imitation Learning with Double Exploration,” Jun. 25, 2025, *arXiv*: arXiv:2506.20307. doi: 10.48550/arXiv.2506.20307.
[24] P. Yu, A. Bhaskar, A. Singh, Z. Mahammad, and P. Tokekar, “Sketch-to-Skill: Bootstrapping Robot Learning with Human Drawn Trajectory Sketches,” Mar. 14, 2025, *arXiv*: arXiv:2503.11918. doi: 10.48550/arXiv.2503.11918.
[25] “NVIDIA Isaac Lab Open-Source Modular Framework,” NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/isaac/lab
[26] “AH V4,” Allegro Hand. [Online]. Available: https://www.allegrohand.com/v4
[27] Arxiv Insights, *An introduction to Policy Gradient methods - Deep Reinforcement Learning*, (Oct. 01, 2018). Accessed: Aug. 19, 2025. [Online Video]. Available: https://www.youtube.com/watch?v=5P7I-xPq8u8
[28] “Multilayer perceptron,” *Wikipedia*. Aug. 09, 2025. Accessed: Aug. 18, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multilayer_perceptron&oldid=1305076934
[29] “PPO — Stable Baselines3 2.7.1a0 documentation.” [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html
[30] “What Is a GPU? Graphics Processing Units Defined.” [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html
[31] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 30, 2017, *arXiv*: arXiv:1412.6980. doi: 10.48550/arXiv.1412.6980.
[32] Y. Huang, K. Xie, H. Bharadhwaj, and F. Shkurti, “Continual Model-Based Reinforcement Learning with Hypernetworks,” Mar. 30, 2021, *arXiv*: arXiv:2009.11997. doi: 10.48550/arXiv.2009.11997.
[33] L. Wang, X. Zhang, H. Su, and J. Zhu, “A Comprehensive Survey of Continual Learning: Theory, Method and Application,” Feb. 06, 2024, *arXiv*: arXiv:2302.00487. doi: 10.48550/arXiv.2302.00487.

[34] A. A. Rusu *et al.*, "Progressive Neural Networks," Oct. 22, 2022, *arXiv*: arXiv:1606.04671. doi: 10.48550/arXiv.1606.04671.
[35] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong Learning with Dynamically Expandable Networks," Jun. 11, 2018, *arXiv*: arXiv:1708.01547. doi: 10.48550/arXiv.1708.01547.
[36] "Self-Evolved Dynamic Expansion Model for Task-Free Continual Learning | IEEE Conference Publication | IEEE Xplore." [Online]. Available: https://ieeexplore.ieee.org/document/10376992
[37] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proc. Natl. Acad. Sci.*, vol. 114, no. 13, pp. 3521–3526, Mar. 2017, doi: 10.1073/pnas.1611835114.
[38] F. Zenke, B. Poole, and S. Ganguli, "Continual Learning Through Synaptic Intelligence," Jun. 12, 2017, *arXiv*: arXiv:1703.04200. doi: 10.48550/arXiv.1703.04200.
[39] A. Mallya and S. Lazebnik, "PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning," May 13, 2018, *arXiv*: arXiv:1711.05769. doi: 10.48550/arXiv.1711.05769.
[40] A. Mallya, D. Davis, and S. Lazebnik, "Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights," Mar. 16, 2018, *arXiv*: arXiv:1801.06519. doi: 10.48550/arXiv.1801.06519.
[41] D. Ha, A. Dai, and Q. V. Le, "HyperNetworks," Dec. 01, 2016, *arXiv*: arXiv:1609.09106. doi: 10.48550/arXiv.1609.09106.
[42] J. von Oswald, C. Henning, B. F. Grewe, and J. Sacramento, "Continual learning with hypernetworks," Apr. 11, 2022, *arXiv*: arXiv:1906.00695. doi: 10.48550/arXiv.1906.00695.
[43] V. K. Chauhan, J. Zhou, P. Lu, S. Molaei, and D. A. Clifton, "A Brief Review of Hypernetworks in Deep Learning," *Artif. Intell. Rev.*, vol. 57, no. 9, p. 250, Aug. 2024, doi: 10.1007/s10462-024-10862-8.
[44] A. Wein, "Introduction to the Low-Degree Polynomial Method".
[45] yahiko, "Introduction à l'interpolation numérique," Developpez.com. [Online]. Available: http://yahiko.developpez.com/tutoriels/introduction-interpolation/
[46] "Piecewise Functions." [Online]. Available: https://www.mathsisfun.com/sets/functions-piecewise.html
[47] M. Buhmann, "Radial basis function," *Scholarpedia*, vol. 5, no. 5, p. 9837, May 2010, doi: 10.4249/scholarpedia.9837.
[48] "Fourier series - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Fourier_series
[49] N. Hager, "An introduction to neural implicit representations with use-cases," Medium. [Online]. Available: https://medium.com/@nathaliemariehager/an-introduction-to-neural-implicit-representations-with-use-cases-ad331ca12907
[50] "Interpolation Using Prelookup - Use precalculated index and fraction values to accelerate approximation of N-dimensional function - Simulink." [Online]. Available: https://www.mathworks.com/help/simulink/slref/interpolationusingprelookup.html
[51] "Gymnasium Documentation." [Online]. Available: https://gymnasium.farama.org/index.html
[52] "Gymnasium Documentation." [Online]. Available: https://gymnasium.farama.org/environments/classic_control/cart_pole.html