Physical Atari: A Robust and Accessible Platform for Real-time Reinforcement Learning on Robots

Khurram Javed[1], **Joseph Modayil**[1,3], **Gloria Kennickell**[1], **Richard S. Sutton**[1,2], **John Carmack**[1]

{kjaved,joseph,gloria,rssutton,johnc}@keenagi.com

[1]**Keen Technologies**
[2]**University of Alberta, Canada**
[3]**The OpenMind Research Institute**

Abstract

We built a robot called *the Robotroller* that actuates an Atari CX40+ controller, and we built a device called *the Atari Devbox* that renders the game frame and the reward signal from the Arcade Learning Environment on a screen. The Robotroller and the Atari Devbox, together with an off-the-shelf camera and a desktop computer, constitute a system that can be used to study reinforcement learning algorithms in the physical world. We call the full system *Physical Atari*.

In this paper, we detail the key decisions that make Physical Atari a robust and accessible platform. To make the system robust, we designed the Robotroller so that all movement is done through bearings, which reduces wear. Additionally, we wrote software that monitors the state of the servos at a high frequency and intervenes to limit forces. Our software plays a similar role to certain reflexes in humans. To make the system accessible, we used affordable off-the-shelf components and parts that can be manufactured using consumer 3D printers. Physical Atari can be built for under $1,000 and has been used for weeks of non-stop reinforcement learning experiments without any mechanical failures. We used it to validate that reinforcement learning algorithms can learn directly on robots, without the need for a simulation or human data, and that even small distribution shifts between learning and deployment can significantly degrade the performance of policies.

1 Real-time Reinforcement Learning

A reinforcement learning agent uses its sensors to observe an aspect of its world, uses the sensory data and its internal state for making decisions, and uses its effectors for influencing the environment with the aim of maximizing the rate of reward that it perceives. Reinforcement learning benchmarks implement the agent-environment interaction as a turn-based game (for example, see Brockman et al., 2016; Bellemare, Naddaf & Bowling, 2023; Beattie et al., 2016), in which the environment waits for the agent's response before changing. This is different from problems in the physical world, in which the state of the environment continues to evolve as the agent makes a decision. We call the setting in which the environment evolves continuously *real-time reinforcement learning*. Real-time reinforcement learning is not a novel setting. For example, Travnik et al. (2018) and Ramstedt & Pal (2019) talk about the setting.

Travnik et al. (2018) proposed a framework called *reactive reinforcement learning* for organizing computation in a real-time reinforcement learning setting. The key idea of their framework is that the agent should execute an action in response to an observation as soon as a choice has been made, and postpone other computational activities, such as learning updates, until after the action has been

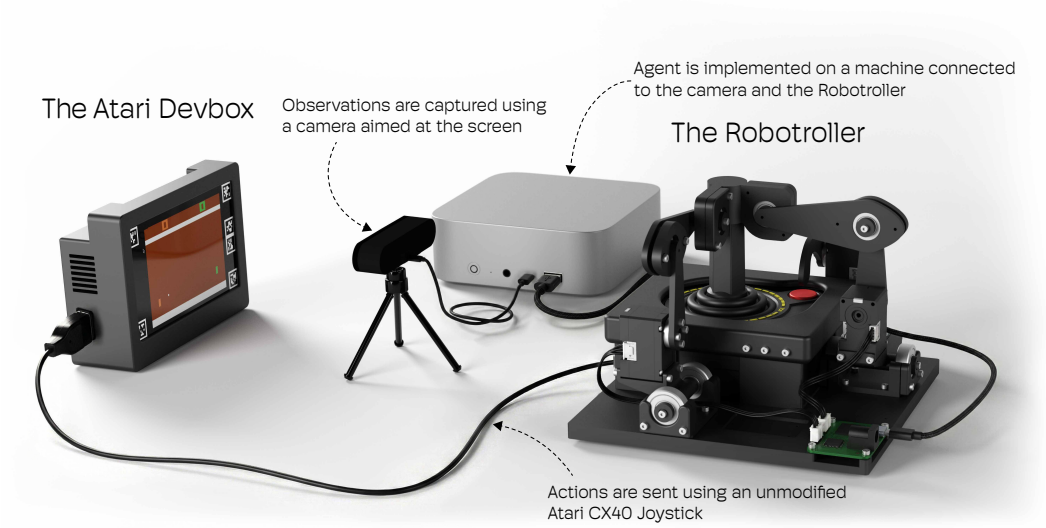

Figure 1: The overview of the Physical Atari platform. On the left is the Atari Devbox—a device that renders Atari 2600 games at 60 FPS, listens to the actions from the CX40+ controller, and displays Apriltags to communicate the reward signal and the location of the game screen. On the right is the Robotroller—a robot that actuates an unmodified CX40+ controller. A reinforcement learning agent captures the screen of the Atari Devbox using a camera and sends commands to the Robotroller to play the game.

executed. Then when it is ready to pick the next action, it should perceive the next observation. This is contrary to the interface used in turn-based RL benchmarks, where the next observation is received in response to the action.

Real-time reinforcement learning poses unique challenges. An agent in a real-time reinforcement learning setting may want to make a decision quickly, to minimize the difference in its perception of the state of the environment and the actual state of the environment. It may want to use different amounts of compute for different decisions, making it possible to take deliberate decisions when the cost of a delay is low and reactive actions when a delay can be catastrophic.

An application of the real-time reinforcement learning setting is robotics. In recent years, reinforcement learning has become a key paradigm for solving challenges in robotics. It has been applied to robotics in different ways. We describe three of them here.

One approach is to create a simulation of the robot and its environment and then use reinforcement learning algorithms to learn policies in the simulated environments (see Abbeel et al., 2007; Akkaya et al., 2019). These policies are then deployed to the robot. A second approach is to get human operators to control the robots and collect trajectories (see Mandlekar et al., 2018; Zhao et al., 2023). These trajectories are then used by an offline-RL algorithm to learn a policy which is deployed to the robot. A third, less common approach, is to learn directly on the robots (see Benbrahim et al., 1992; Haarnoja et al., 2018; Smith et al., 2022; Wu et al., 2023). The advantages of learning directly on the robots are that it does not require a simulator or human data and there is no distribution shift between the learning and deployment environments. The focus of this paper is the third approach.

There is no platform for studying reinforcement learning directly on robots—the third approach mentioned above—that is both reliable and accessible. We wanted to build one such platform. We built a platform called *Physical Atari* that is *accessible*, *reliable*, *easy to use*, and *versatile*. Here accessible means that it is built using affordable and easily available components, reliable means that it can be used for long experiments without mechanical failures, easy to use means that it can be used for long periods without requiring interventions, and versatile means that it consists of a wide range of tasks.

The two key components of Physical Atari are *the Robotroller* and *the Atari Devbox*. The Robotroller is a robot that actuates an unmodified commercially available Atari CX40+ controller. The Atari Devbox is a device with a 5-inch display that renders the game screen and the reward signals of games from the Arcade Learning Environment (ALE) (Bellemare et al., 2013).

We picked games from ALE as the tasks for our platform because the performance of reinforcement learning algorithms in simulation on these games is well understood, and these games have proven useful for reinforcement learning research. ALE has been used in several seminal works, such as DQN (Mnih et al., 2013), Rainbow DQN (Hessel et al., 2018), MuZero (Schrittwieser et al., 2020), Data-efficient Rainbow (Van Hasselt et al., 2019), and BBF (Schwarzer et al., 2023). Physical Atari brings the richness of the ALE benchmark to the physical world and can be used to study challenges, such as time delays and non-stationarities, that are ignored in simulation.

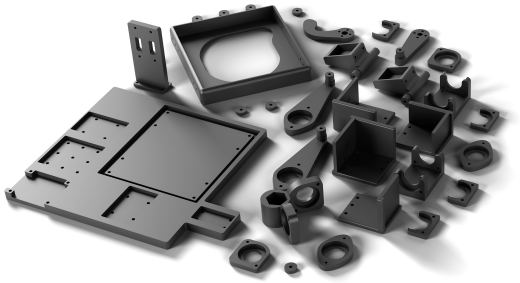

Figure 2: The components of the Robotroller that are 3D printed. On a consumer Bambu Lab P1S printer, all the parts can be printed in around 12 hours.

2 The Physical Atari Platform

Figure 1 shows an overview of the Physical Atari platform. The platform consists of a camera that is pointed at the Atari Devbox, a computer that is connected to the camera, the Robotroller that is connected to the computer, and a CX40+ controller that is connected to the Atari Devbox.

The computer uses the camera as its sensor, and it sends one of the eighteen discrete actions used by ALE to the Robotroller. The actions are executed by moving the servo motors to preprogrammed positions.

The camera is the Razer Kiyo Pro, which is one of the few off-the-shelf cameras capable of streaming uncompressed video. Using an uncompressed data stream reduces latency. The camera is connected to the computer using a USB 3.1 connection. We used a Framework desktop with an AMD Ryzen AI Max+ 395 chip as our computer; other computers that support USB 3.1 should work fine.

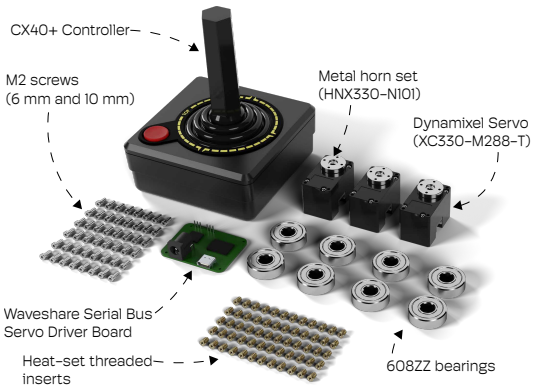

Figure 3: The components of the Robotroller that have to be bought. These include screws, bearings, servos, electronics, and threaded inserts. The total cost of the parts is around $400.

2.1 The Robotroller

The Robotroller consists of two types of parts. The first type is custom parts designed in CAD software (Autodesk Fusion) that have to be manufactured. These are shown in Figure 2. We designed these parts so they can be printed on a consumer 3D printer, like the Bambu Lab P1S, using a cheap filament, like PLA. The second type is parts that have to be bought. These include screws, bearings, servos, threaded inserts, and electronics. These are shown in Figure 3. The fully assembled Robotroller is shown in Figure 4.

A key goal when designing the Robotroller was to ensure it can run for long periods without significant wear and tear. To achieve this goal, we used ball bearings in all moving parts of the system. We used 608ZZ bearings that are readily available at hardware stores. Using ball bearings reduces

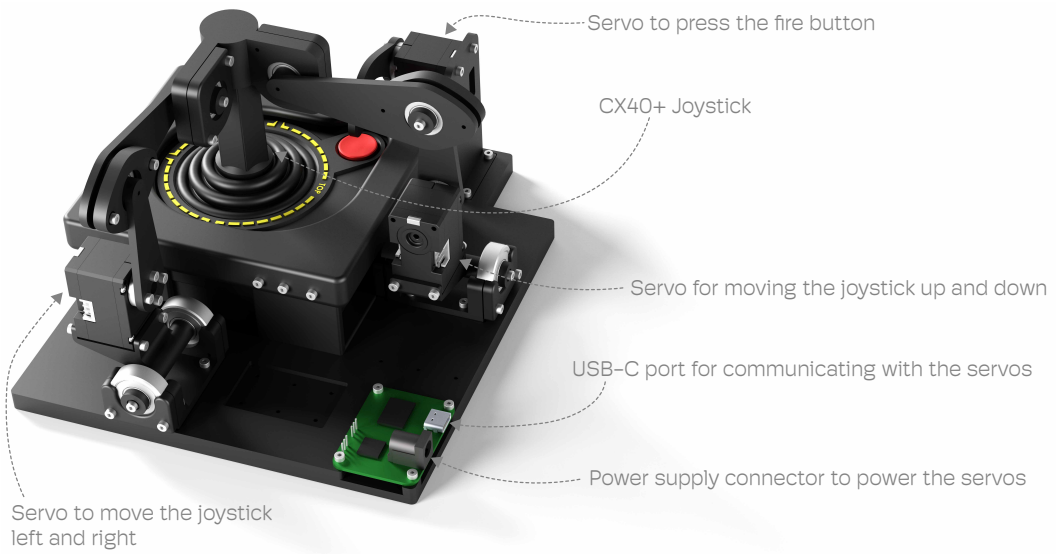

Figure 4: The Robotroller is a robot that reliably actuates an unmodified CX40+ controller using three servo motors. It uses the Dynamixel XC-330 servo motors and a Waveshare PCB to communicate with the servo motors over a serial connection via a USB-C port. To minimize wear, the Robotroller uses 608ZZ bearings in all moving parts.

friction, which reduces wear that is typically associated with 3D-printed PLA parts. To transfer the torque of the servos to the joystick through ball bearings, we positioned the two servos that control the joystick so they can rotate about two axes that meet at the pivot point of the joystick. This configuration is illustrated in Figure 5.

When testing the Robotroller, we discovered that some screws would eventually come loose after extended use. We fixed this by using a thread-locking compound when assembling the Robotroller. We used *Loctite 243 Threadlocker*.

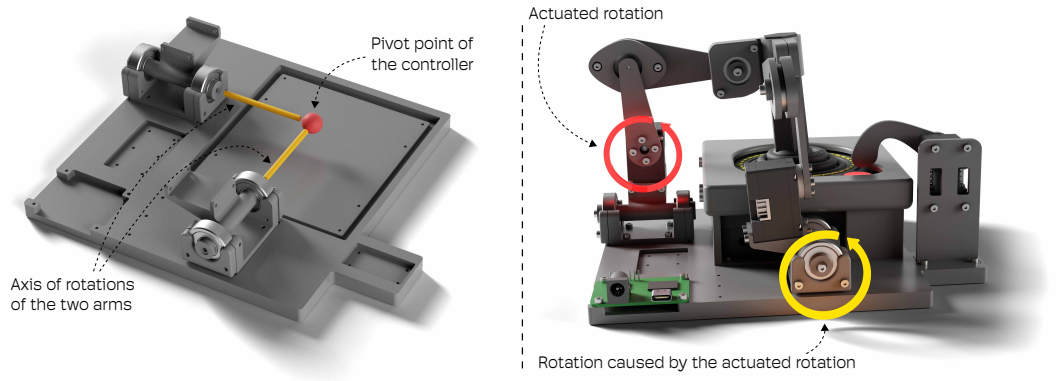

Figure 5: The key to making the Robotroller reliable is the design that uses bearings for all movements. This was a challenge to implement because as one arm of the robot pulls or pushes the joystick, the other arm has to move to accommodate the movement of the joystick. We deal with this by putting both arms on bearings whose axes of rotation are aligned with the pivot point of the joystick. The two axes meet exactly at the pivot point, as shown in the figure on the left. The figure on the right shows how rotation of one servo (axis of rotation shown as a red circle) causes the other arm to tilt (axis of rotation shown as a yellow circle).

After further testing we discovered that the internal gears of the servos would wear out after extended use. This happened because in the initial design, we used the cheaper Dynamixel XL-330 servos, which use gears made of engineering plastic. We replaced the servos with the more expensive Dynamixel XC-330 servos that use metal gears. In our extended tests lasting several weeks of continuous use, none of our robots showed signs of wear in the metal gears.

The XC-330 servo uses metal gears but its horn—the part used to connect the servo to external components—is made of engineering plastic. We found that in one build, this part broke. Dynamixel sells a metal replacement (HNX330-N101) for this part. We replaced the plastic horns with the metal horns on all three servos, which fixed the issue.

Once we made the Robotroller reliable, we noticed that the CX40+ controller wore out after extended use. This was surprising because the controller is used by people for long periods of time without issues. We traced the issue to the motion profile of the Robotroller, which was too aggressive and put unnecessary stress on the controller.

The servos use a PID controller for position-based control. The setpoints for positions are set manually so the servos can move to trigger any one of the eighteen actions of ALE when given a command. The PID controller has three key parameters—P, I, and D. By default, Dynamixel ships the servos with a large value of the P parameter, and zero values for the I and D parameters. We noticed that the large value of the P parameter created jerky movements, which likely put too much stress on the controller.

We made the motion of the servos smoother by changing the values of the PID controller parameters. We used a small value of the P parameter, a small value of the D parameter, and a large value of the I parameter. The exact values are in Appendix E, and an anonymized video showing the movement of the Robotroller before and after tuning is here.

The new values of the PID controller introduced another problem. If, for some reason, a servo did not reach its setpoint quickly, then the current would spike to very large values and put the servo in a state that required a manual reboot. This was because of the large value of the I parameter, which integrates the error over time to generate a control signal. We fixed this by introducing a *high-current reflex*—a preprogrammed behaviour in the servos that is triggered if the current exceeds a certain threshold.

The high-current reflex is implemented in a high frequency loop that continually checks the value of the currents in the servos. If at any point the current exceeds a threshold, the goal position of the servo is set to its current position, and its torque is disabled for one millisecond. This reflex acts as a safety mechanism to protect the servos from damaging themselves or putting them in a state that requires a manual reboot. We call this policy a reflex because there are numerous examples of reflexes in the human body that protect the body from damage in a similar way.[1]

After updating the PID parameters and adding the high-current reflex, we used the Robotroller continuously for weeks with the same CX40+ controller without signs of significant wear. We provide more details of the Robotroller in Appendix B, and the complete step-by-step instructions for building the Robotroller are available at an anonymous link here.

2.2 The Atari Devbox

The Atari Devbox is a compact device that uses a Raspberry Pi 5 as its compute unit and a 5-inch Waveshare display. We designed a custom chassis in Autodesk Fusion that can be 3D-printed to house the components. Since the Raspberry Pi 5 lacks a built-in DB9 port required to connect the Atari CX40+ joystick, we connected a male DB9 connector directly to the GPIO pins of the Raspberry Pi 5. This connector is exposed through the chassis of the Atari Devbox. A picture of the Atari Devbox is shown in Figure 6.

[1] An example is the Golgi tendon reflex, a reflex to protect against too much tension on the tendons and muscles of the body.

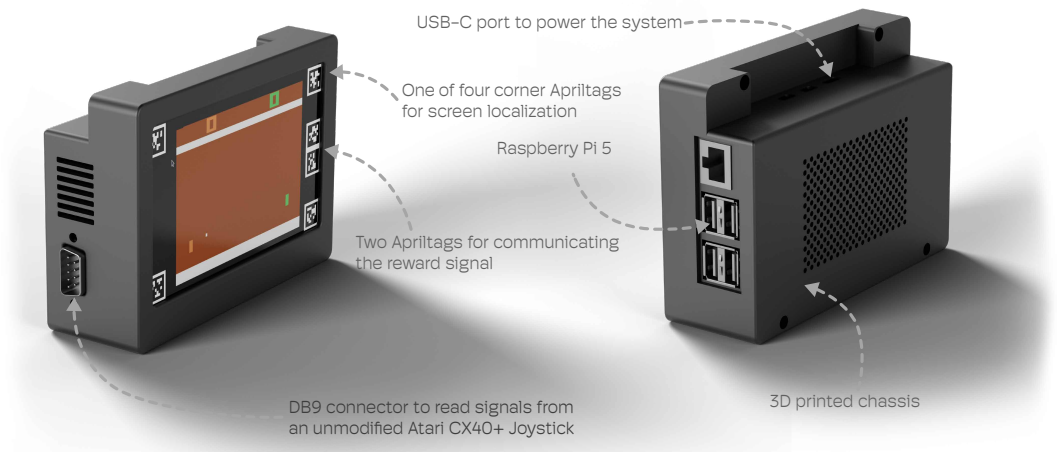

Figure 6: The Atari Devbox is a device that renders Atari 2600 games at 60 FPS, listens to the signals from the CX40+ controller, and displays Apriltags to communicate the reward signal and the location of the game screen. It uses Arcade Learning Environment (ALE) running on a Raspberry Pi 5. To read signals from the CX40+ controller it connects the controller to the GPIO pins of the Raspberry Pi 5. It uses an off-the-shelf Waveshare Display that connects to the Raspberry Pi 5 using its SPI interface. A C++ program running on the Raspberry Pi 5 reads the signals from the GPIO pins at a very high frequency and converts them to one of the 18 actions of the Arcade Learning Environment.

We wrote a program that reads the state of the GPIO pins with sub-millisecond latency and maps the state of the five wires of the CX40+ controller (Up, Down, Left, Right, Fire) to the 18 discrete actions of the Arcade Learning Environment (ALE). The program then passes the action to the ALE environment and renders the resulting game frame to the screen at 60 FPS.

Beyond the game screen, the program renders a set of AprilTags (Olson, 2011): four corner tags to help an agent localize the game screen from camera images, and a dynamic set of tags on one side of the screen to encode the reward signal.

The complete step-by-step instructions for building the Atari Devbox are available here and more details about the Atari Devbox are in Appendix A.

3 Characterizing the Response Time of Physical Atari

A physical system has a delay between receiving a stimulus and responding. Rendering a frame to the screen, capturing the screen with a camera, processing the image to get an action, and moving the controller to execute the action all take time.

To quantify this delay, we developed a calibration tool that measures the end-to-end response time of the entire system. The tool renders a visual marker—AprilTags with IDs from 0 to 17—on the screen to command a specific action. As it picks a visual marker to render, it also starts a high-resolution timer. The robot, perceiving this marker through its camera, actuates the joystick to perform the commanded action. The tool detects the physical actuation by monitoring the electrical signals on the controller's wires. As soon as it has read the commanded action, it stops the timer.

This process measures the total response time of the system, encompassing the time taken for display rendering, camera capture, image processing, and mechanical actuation. We report the response time for various action pairs in Figure 7. Each reported measurement is the average of 30 trials. The response time is roughly 165 ms, which is comparable to typical human reaction times. This response time does not include the time it would take to pick an action using a deep neural network.

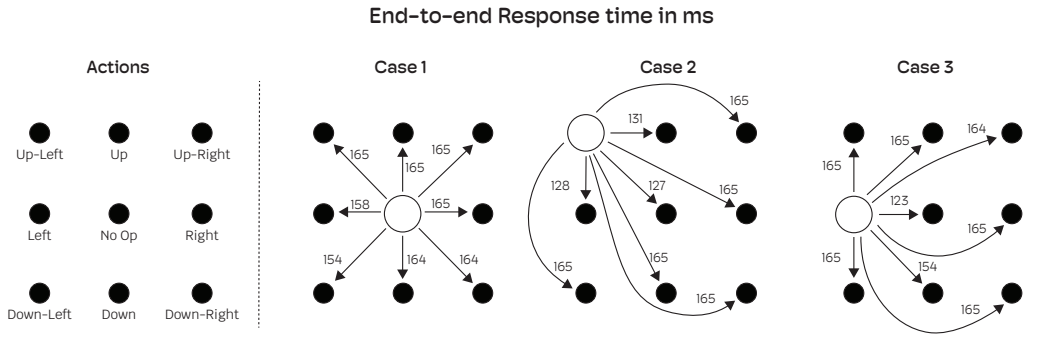

Figure 7: We measured the end-to-end response time of the Physical Atari platform. The response time is the total time required to render on the screen, capture the image with the camera, transfer the camera image to the computer, and move the joystick to the desired action. We plot the times in milliseconds in the above plot from the center (Case 1), top-left (Case 2), and left (Case 3) positions. The latency of the system is roughly 165 ms, which is comparable to typical human reaction times.

4 Experimental Setup: Network Architecture and the Learning Algorithm

We used the Physical Atari platform with a deep reinforcement learning algorithm implemented in the Reactive Reinforcement Learning framework (Travnik et al., 2018). This means that our agent sends the action to the Robotroller as soon as it is done picking the action. The learning update happens after the action has been sent and before the next observation has been perceived.

The agent uses a fully convolutional network that progressively downsamples its visual input. Each camera frame is resized to 128x128x3 and blended into an exponential moving average before the network receives it. The agent then normalizes each smoothed observation by subtracting its mean pixel value and dividing by its standard deviation.

The smoothed observation passes through a sequence of 3x3 convolutional layers (LeCun et al., 1989). After each convolution, except the final one, the feature maps are downsampled with a 3x3 max-pooling operation with stride 2 and then passed through a ReLU nonlinearity (Nair & Hinton, 2010). This process continues until the spatial resolution is reduced to 2x2, and a linear layer maps the final feature map to a scalar value. The first convolutional layer has 24 filters, and we double the number of filters after each convolution.

A distinguishing feature of this architecture is the late fusion of the agent's action history. The agent encodes its recent actions as a vector and uses an exponential moving average of that vector to summarize recent control decisions. This gives the network a compact description of what the agent has been doing over the recent past.

For Atari, we represent each action with a factored binary code. This code separates the action into vertical movement, horizontal movement, and fire, so related actions can share structure in their representation. The first three bits encode vertical movement, the next three bits encode horizontal movement, and the final two bits encode whether the fire button is pressed. For example, `UP` is encoded as `[1,0,0, 0,1,0, 1,0]` because it selects up, no horizontal movement, and no fire. `UPRIGHT` is encoded as `[1,0,0, 0,0,1, 1,0]` because it keeps the same up and no-fire components but changes the horizontal component from neutral to right.

We combine this action history with the visual stream after the first convolutional stage and immediately before the second convolution. Each entry of the action vector is copied to every spatial location in the current convolutional representation, which produces one constant feature map per action component. We then concatenate these feature maps with the visual feature maps along the channel dimension.

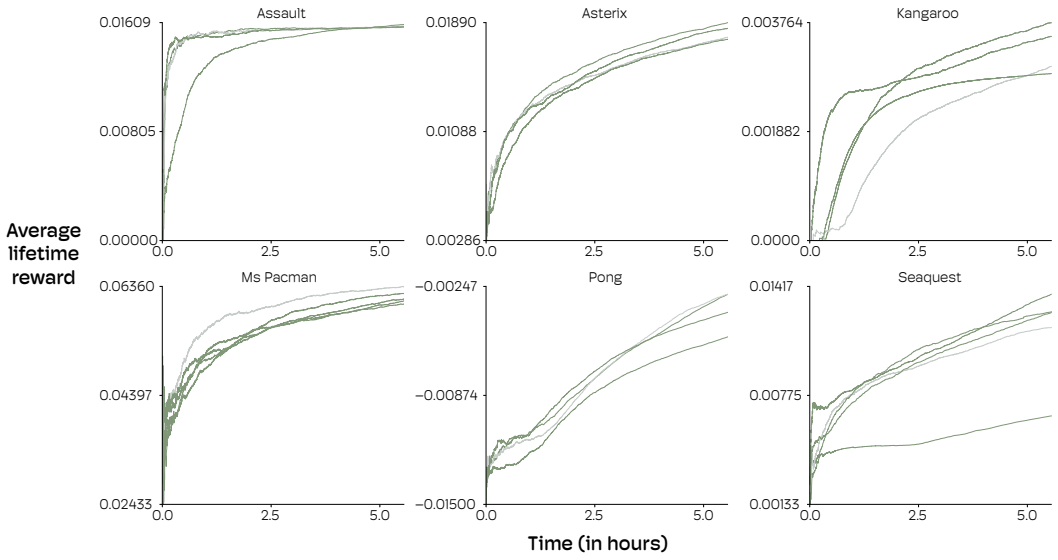

Figure 8: Average reward over time during real-time reinforcement learning on six Atari games. At every point in time the y-axis plots the cumulative reward divided by total steps. The agent runs at 30 frames per second. On each game, we repeated the experiment 4 to 5 times with the same hyperparameters. Collectively, these experiments took around 145 hours and required no intervention.

This design allows each convolutional filter to condition on the same recent-action summary at every spatial location. In this respect, the architecture differs from DQN (Mnih et al., 2013) and its variants, where the network outputs a value for each action instead of receiving the action history as part of its input. As a result, our value estimator is explicitly conditioned on both a temporally smoothed visual observation and the recent sequence of actions.

The targets are constructed using n-step returns (Sutton, 1988; Sutton & Barto, 2018). The agent uses an ϵ-greedy behavior policy with a fixed value of ϵ.

The agent also maintains a separate target network. It uses this target network to evaluate actions when it selects behavior and to compute the bootstrap value at the end of each n-step return. We update the target network by copying the online network into it at a fixed period, which stabilizes learning by keeping the target values more slowly changing.

The agent learns using a sample of data from prior experience stored in a replay buffer. Instead of sampling uniformly from the full buffer, it reserves a fraction of the batch for the most recent online transitions and samples the remaining elements of the batch randomly from the buffer. The effectiveness of this modification has been shown by Zhang & Sutton (2018).

The agent updates its weight parameters to minimize the mean squared error between the network's predictions and the multi-step return targets. It uses AdamW (Kingma & Ba, 2015; Loshchilov & Hutter, 2019) to update the parameters in the convolutional layers, and uses SGD with momentum to update the parameters in the final linear layer. We provide more details of the experiment setup and the hyperparameters used in the experiments in Appendix D.

The performance of the agent is measured as the average reward over all past experience. That is, if r_i is the scalar reward at the ith time step, then the average reward up to time step t is

$$\frac{1}{t}\sum_{i=1}^{t} r_i.$$

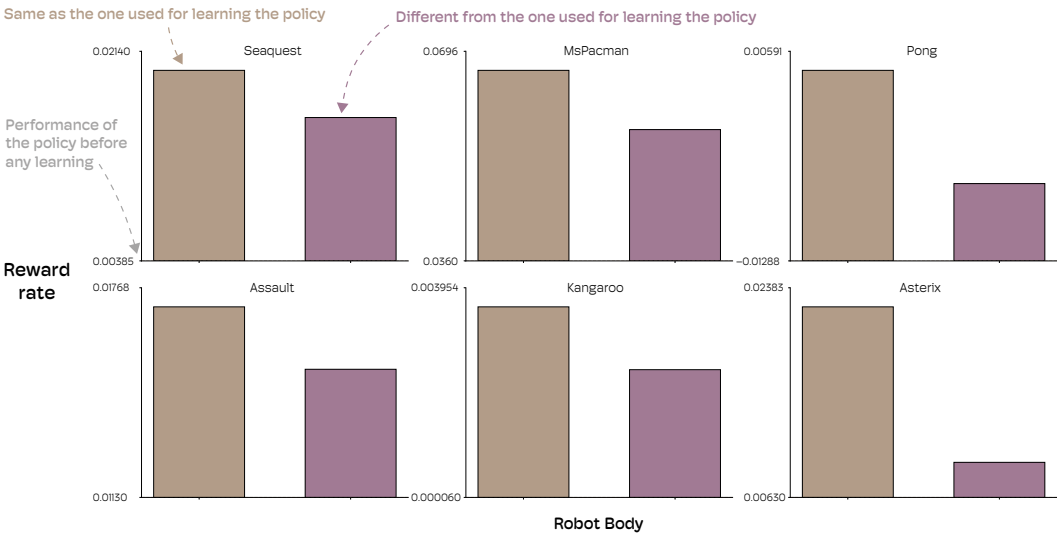

Figure 9: We evaluated policies that learned with 6 hours of experience twice, once on the robot body used for learning and once on a different but identically built body. On every game, the performance was better when tested on the body used for learning. This shows that even very small distribution shifts can negatively impact performance and learning directly on every individual robot body can be advantageous.

We did not use episodic returns as an evaluation metric because average reward is a better metric for continual reinforcement learning (Naik & Sutton, 2019).

5 Experiment Results

We conducted two experiments to characterize the performance of the Physical Atari platform. The first experiment tests whether existing reinforcement learning algorithms can consistently learn on the platform without mechanical failures or human intervention. The second experiment tests how well policies learned on one Robotroller transfer to a different Robotroller.

5.1 Physical Atari can be used for Weeks of Non-Stop Learning Without Mechanical Failures

We ran our agent on the Physical Atari platform to learn to play six games—Pong, Seaquest, MsPacman, Assault, Asterix, and Kangaroo—for five and a half hours. For each game, we repeated the experiment at least 4 times and report the results in Figure 8.

The agent learned on all games, and its performance was consistent across multiple runs. These experiments took nearly 145 hours and required no interventions. An anonymized video of the system learning is here.

5.2 Small Distribution Shifts can have Large Negative Impacts on Performance in Physical Atari

An appeal of the Physical Atari platform is that it can be used to study questions that often arise in robotics. One such question is how small differences in different robot bodies impact performance. We used our setup to answer this question. We learned on six games for six hours and stored the policies. We then evaluated these policies twice, once using the Robotroller and controller that was used for learning the policies, and once using a different Robotroller and controller. We show the results in Figure 9.

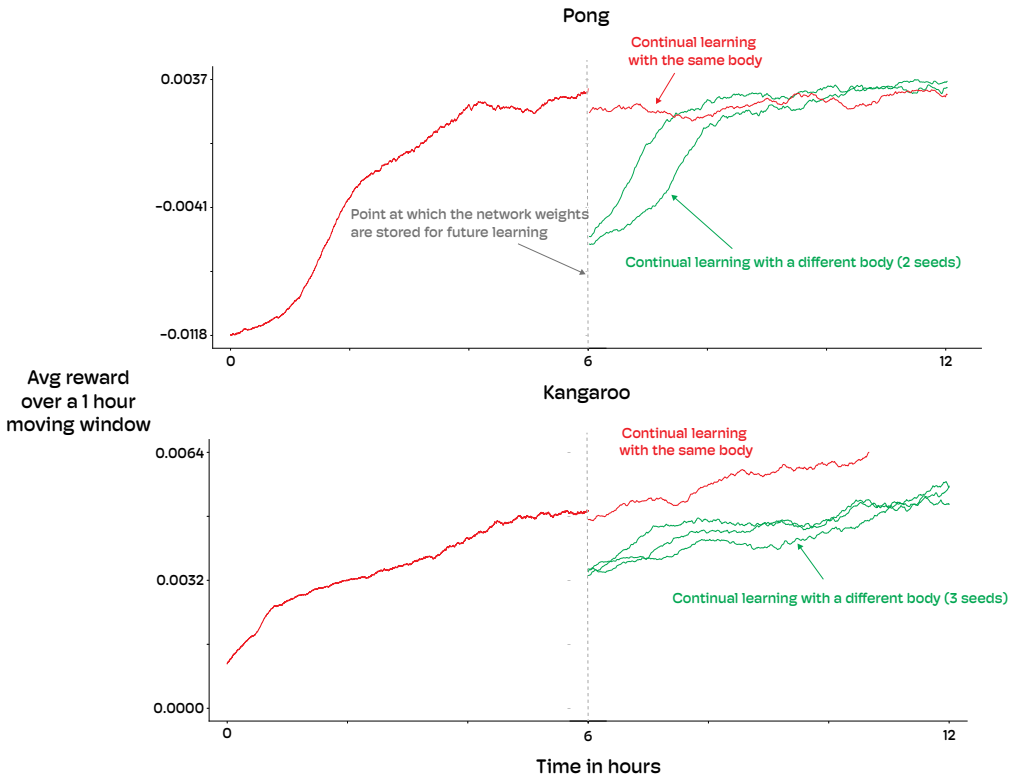

Figure 10: In another experiment, we switched the body of the robot after 6 hours of learning. The switch creates a distribution shift if the two bodies are not exactly identical. We then let the agent continue to learn with the new body. Switching the bodies degraded the performance of the policies and letting the agent learn continually improved their performance.

Across all evaluated games, the policies consistently performed worse when tested on the Robotroller that was not used for learning. The difference was largest in games that required timing actions precisely. For example, we noticed that in Pong the policy deployed on the different body moved the paddle in the right direction toward the ball but it consistently missed the ball by a small margin. Even a very minute mismatch in timing can be hugely detrimental to performance in Pong.

The degradation of performance of policies when deployed on a different body shows that even minimal distribution shifts between learning and deployment—arising from slight, inevitable physical differences between robots—can significantly impair the agent's capabilities.

We did one more experiment on two games. After deploying the policy on a new body, we let our learning algorithm continue to adapt. We show the results in Figure 11. Learning continually on the new body improved the policy's performance.

6 Conclusions and Future Work

The *Physical Atari* platform is the first robotic platform that is both affordable and can be used to run experiments for weeks without requiring human intervention.

One exciting use of this platform would be to compare the three popular paradigms of RL in robotics—learning in simulation, learning from human data, and learning directly on the robots—in a systematic way. Our experiments show that even very small discrepancies between learning and deployment, introduced by using two different bodies, can lead to large differences in performance. Learning in simulation introduces even larger discrepancies between learning and deployment. It

would be interesting to see how well policies trained for a long time in simulation—for hundreds of millions of frames—perform when tested on the physical system. Similarly, it would be interesting to see how policies learned using a few hours of human data compare to policies learned purely from experience.

The effectiveness of the high-current reflex is another interesting result that provides a path to building more robotics platforms that are suitable for real-time reinforcement learning. Augmenting existing robotics platforms with reflexes so that a reinforcement learning agent cannot damage them could make them more suitable for real-time reinforcement learning.

References

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. *Advances in Neural Information Processing Systems*, 19.

Akkaya, I., Andrychowicz, M., Chociej, M., Chiek, M., Boby, A., Baker, B., ... & Zaremba, W. (2019). Solving Rubik's cube with a robot hand. *arXiv Preprint arXiv:1910.07113*.

Benbrahim, H., Doleac, J., Franklin, J., & Selfridge, O. (1992, June). Real-time learning: A ball on a beam. In *International Joint Conference on Neural Networks*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. *arXiv Preprint arXiv:1606.01540*.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., ... & Petersen, S. (2016). DeepMind Lab. *arXiv Preprint arXiv:1612.03801*.

Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., & Levine, S. (2018). Learning to walk via deep reinforcement learning. *arXiv Preprint arXiv:1812.11103*.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*.

Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. *International Conference on Learning Representations*.

Mandlekar, A., Zhu, Y., Garg, A., Booher, J., Spero, M., Tung, A., ... & Fei-Fei, L. (2018, October). RoboTurk: A crowdsourcing platform for robotic skill learning through imitation. In *Conference on Robot Learning*. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *arXiv Preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*.

Naik, A., Shariff, R., Yasui, N., Yao, H., & Sutton, R. S. (2019). Discounted reinforcement learning is not an optimization problem. *arXiv Preprint arXiv:1910.02140*.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*.

Olson, E. (2011, May). AprilTag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*.

Ramstedt, S., & Pal, C. (2019). Real-time reinforcement learning. *Advances in Neural Information Processing Systems*, 32.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., ... & Silver, D. (2020). Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*.

Schwarzer, M., Obando-Ceron, J., Courville, A., Bellemare, M. G., Agarwal, R., & Castro, P. S. (2023). Bigger, better, faster: Human-level Atari with human-level efficiency. In *Proceedings of the 40th International Conference on Machine Learning*.

Smith, L., Kostrikov, I., & Levine, S. (2022). A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv Preprint arXiv:2208.07860*.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

Travnik, J. B., Mathewson, K. W., Sutton, R. S., & Pilarski, P. M. (2018). Reactive reinforcement learning in asynchronous environments. *Frontiers in Robotics and AI*.

Van Hasselt, H. P., Hessel, M., & Aslanides, J. (2019). When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*, 32.

Wu, P., Escontrela, A., Hafner, D., Abbeel, P., & Goldberg, K. (2023, March). Daydreamer: World models for physical robot learning. In *Conference on Robot Learning*. PMLR.

Zhao, T. Z., Kumar, V., Levine, S., & Finn, C. (2023). Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv Preprint arXiv:2304.13705*.

Zhang, S., & Sutton, R. S. (2017). A deeper look at experience replay. *arXiv Preprint arXiv:1712.01275*.

Supplementary Materials

The following content was not necessarily subject to peer review.

A Atari Devbox Technical Details

The Atari Devbox serves as the interface between the physical world and the emulated Atari environment. It is designed to be low-latency, reliable, and reproducible using off-the-shelf components.

A.1 Software Architecture

The core software is a C++ application running on a Raspberry Pi 5 with Debian 12 (Bookworm). It integrates several key libraries to ensure high performance. The Arcade Learning Environment (ALE) provides the underlying Atari 2600 emulation, while SDL2 handles high-performance 2D rendering of the game frame and AprilTags. To ensure minimal input lag, the application uses 'libgpiod', a fast, modern C++ interface for reading GPIO pins. Additionally, Dear ImGui is used for debugging overlays. The main loop runs at 60 Hz, synchronized with the game emulation. In each cycle, the system reads the GPIO pins, steps the ALE environment, and renders the new frame and auxiliary tags. The GPIO pins are read just before sending the action to the ALE emulator. This assures that the system uses the latest action.

A.2 GPIO Interface and Action Mapping

The Atari CX40+ joystick is a device with five wires connected to a female DB9 connector. We connect these wires to the Raspberry Pi's GPIO pins (see Table 1). The internal pull-up resistors are enabled, so a button press pulls the corresponding pin to ground (logic 0). The raw GPIO states are mapped to the 18 discrete actions defined by the ALE. For example, if both "Up" (Pin 17) and "Right" (Pin 24) are active, the system sends the `PLAYER_A_UPRIGHT` action to the emulator. This allows the physical joystick to access the full action space of the Atari 2600.

Table 1: GPIO Pin Mapping for CX40+ Controller

Function	GPIO Pin (BCM)
Up	17
Down	27
Left	22
Right	24
Fire	23

A.3 Visual Interface and Reward Communication

To enable the agent to perceive the environment and reward solely through vision, the Atari Devbox renders a composite image containing the game screen and several AprilTags. Four static AprilTags (IDs 0–3) are rendered at the corners of the screen to allow the agent to compute a homography and extract the game frame, regardless of the camera's precise position or angle. Since the agent observes the world through a camera, the reward signal must be encoded visually. We use two dynamic AprilTags positioned on the right side of the screen to transmit reward information. A Value Tag at the top indicates the value of the reward: ID 10 for zero, ID 11 for 1, and ID 12 for -1 rewards. A Change Indicator Tag at the bottom distinguishes between consecutive rewards by cycling through IDs 15, 16, and 17 every time a non-zero reward is received. If the reward is zero, this tag remains static. This state-change mechanism ensures that the agent can accurately count the number of reward events even if they occur in rapid succession.

B Robotroller C++ Library

To enable high-performance control of the Robotroller, we developed a C++ library with Python bindings. This library provides a unified interface for camera capture, robot control, and environment interaction.

B.1 Core Components

The library consists of three main modules working in tandem. One module handles video capture from the webcam in a separate thread to ensure the most recent frame is always available, minimizing latency, while supporting configuration of parameters like focus and exposure. A second module interfaces with the Dynamixel servos, mapping the 18 discrete Atari actions to specific servo positions using a PID controller built into Dynamixels and safety features like current monitoring. Finally, a third system utilizes the AprilTag library to detect tags in the camera feed, serving two critical functions: image rectification to crop the game screen, and reward extraction to decode the visual reward signals.

B.2 Environment Interface

The library integrates these components into a cohesive environment, exposing a `send_action` method and a `perceive` method. The `send_action` method sends the specified action to the Robotroller to trigger servo movement. The `perceive` method captures the latest frame from the camera and detects tags to rectify the image and extract the reward. This design abstracts away the complexity of the physical hardware, allowing reinforcement learning agents to interact with the Physical Atari platform using a simple API.

B.3 High-current Reflex

To ensure long-term reliability and prevent mechanical failure, the library implements a real-time safety reflex within the servo control loop. This mechanism protects against overcurrent events, which typically indicate that the robot is stalled or pushing against a hard limit. An overcurrent event can also put the Dynamixel into a state that requires a manual reboot of the servos. The following logic is executed continuously:

```
for (auto servo : list_of_servos) {
    int16_t cur = servo->getPresentCurrent();
    if (std::abs(cur) > 1200) {
        // Stop movement by setting target to current position
        servo->setPosition(servo->getPresentPosition());

        // Cycle torque to release mechanical tension
        servo->disableTorque();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        servo->enableTorque();
    }
}
```

When the current magnitude exceeds the threshold of 1200 mA, the system sets the target position to the servo's current position to halt movement and momentarily toggles the torque.

C Experiment Harness

To conduct reproducible experiments, we developed a unified Python harness. This harness orchestrates the interaction between the agent and the physical environment, manages data logging, and

handles experiment configuration. It interfaces directly with the hardware using the Python bindings to the C++ library described in Appendix B.

C.1 Training Loop

The core training loop is designed to minimize latency and ensure consistent timing. In each iteration, the harness first calls `env.perceive()` to update the agent state from the sensors (camera and AprilTags). It then retrieves the latest observation and reward. The agent's `get_action` method is called with the current observation to select the next action, which is immediately sent to the environment via `env.send_action(action)` to trigger actuation. Finally, the agent's `learn` method is invoked to update its policy based on the transition.

C.2 Data Logging and Reproducibility

The harness automatically logs comprehensive experiment data to a MongoDB database. All hyperparameters, random seeds, and environment settings are stored with the results. Training metrics, including average reward and episode scores, are tracked in real-time. Additionally, model weights and optimizer states are serialized and stored using GridFS, enabling exact resumption of experiments and post-hoc analysis. We will open-source the harness and the raw data in the final version of the paper.

D Hyperparameters

Table 2 lists the hyperparameters used for the experiments. All games use the same set of hyperparameters.

Table 2: Hyperparameters used in the experiments.

Hyperparameter	**Value**
Observation Size	128×128
Observation Channels	3
Base Channels	24
Policy Skip	2
Train Skip	4
Observation EMA ($2^{\log_2}$)	2^{-3}
Momentum	0.9
Learning Rate ($2^{\log_2}$)	2^{-14}
Linear Learning Rate ($2^{\log_2}$)	2^{-17}
Exploration ($2^{\log_2}$)	2^{-6}
Target Model Update Period ($2^{\log_2}$)	2^{5}
Train Batch Size	16
Train Repetitions	1
Multi-step Returns (n)	12
Discount Factor (γ)	0.99
Online Samples	4
Action Encoding	1
Action EMA Rate	0.875
Action Steps	4
Action Layers	2
Max Frames Without Reward	18000
Total Steps	600000
Evaluation Steps	50000
FPS	30

Figure 11: A photo of three Physical Atari setups.

The hyperparameters dictate how the agent perceives the environment, processes actions, and updates its policy. The **Observation Size** of 128×128 defines the resolution to which the input frames are downsampled. To smooth visual inputs and reduce noise, an exponential moving average (EMA) is applied to the observations with a rate determined by **Observation EMA** (2^{-3}). Similarly, the agent maintains an EMA of its action history using the **Action EMA Rate** (0.875). This action history is injected into the convolutional network at specific depths defined by **Action Layers** (bitmask 2), allowing the agent to condition its value estimates on recent decisions. The **Action Encoding** parameter (1) specifies that actions are represented using a factored binary encoding rather than a one-hot vector.

The learning process is governed by n-step returns, where n is set to 12, meaning the target value aggregates rewards over the next 12 frames before bootstrapping. Future rewards are discounted by a **Discount Factor** of 0.99 per frame. The agent updates its weights every **Train Skip** (4) frames using a **Train Batch Size** of 16. Crucially, to ensure the agent learns from its most immediate experiences, **Online Samples** (4) of the batch are forced to be the most recent transitions, while the rest are sampled uniformly from the replay buffer.

The convolutional layers are updated using AdamW with a **Learning Rate** of 2^{-14}, while the final linear layer is updated using SGD with **Momentum** (0.9) and a smaller **Linear Learning Rate** of 2^{-17}. A target network is hard-updated every **Target Model Update Period** ($2^5 = 32$) frames. Finally, exploration is managed via an ϵ-greedy strategy with a fixed ϵ of 2^{-6} (**Exploration**), and new actions are selected every **Policy Skip** (2) frames.

E Robotroller and Camera Hyperparameters

This section details the specific configuration used for the Robotroller and the camera in our experiments.

E.1 Camera Configuration

The camera is configured with manual focus and exposure to ensure consistent visual observations across frames, preventing auto-adjustment artifacts. The camera captures frames at a resolution of (1280×720). Table 3 lists the specific parameters.

Table 3: Camera Configuration Parameters.

Parameter	Value
Camera Resolution	1280×720
Camera FPS	30
Camera Focus	370 (Manual)
Camera Zoom	100
Camera Exposure	20 (Manual)
Camera Brightness	128
Camera Contrast	128

E.2 Robot Configuration

The Robotroller's movement is governed by a set of parameters that control communication and motor behavior. The **Baud Rate** of 1,000,000 sets the speed of serial communication between the computer and the Dynamixel servos.

The servos use a Proportional-Integral-Derivative (PID) controller to precisely reach and hold target positions. The **P Parameter** (1500) determines how aggressively the motor reacts to position error; a higher value increases speed but may cause overshoot. The **I Parameter** (6000) corrects for steady-state error, ensuring the joystick is held firmly against resistance. The **D Parameter** (1500) dampens the movement by reacting to the rate of change, reducing oscillation and ensuring smooth motion.

The servo positions are calibrated specifically for the robot to map the discrete Atari actions to precise physical encoder values (0-4095). **Noop Position (Default)** (2048) represents the neutral position. **Right/Left Position** and **Up/Down Position** define the target values for pushing the joystick in the respective directions. **Button Default Position** (2000) and **Button Pressed Position** (1932) define the released and actuated states of the fire button, respectively. Table 4 lists these parameters.

Table 4: Robotroller Configuration Parameters.

Parameter	Value
Baud Rate	1,000,000
P Parameter	1500
I Parameter	6000
D Parameter	1500
Noop Position (Default)	2048
Right Position	2130
Left Position	1925
Up Position	2180
Down Position	1960
Button Default Position	2000
Button Pressed Position	1932