Learn To Race - Distributed Reinforcement Learning and Optimization Thrust

Shijie Bian
Mentors: **Arav Agarwal** **Jonathan Francis**
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{sbian,arava,jmf1}@cs.cmu.edu

Abstract

This report delves into the research thrust of distributed reinforcement learning (RL) and optimization within the Learn to Race (L2R) project. Our focus centers on propelling autonomous racing agents by leveraging distributed training to expedite the learning process. L2R integrates a multimodal environment, utilizing diverse sensors and computer vision encoders to empower autonomous agents in real-time vehicle control and environment perception. A pivotal innovation in our approach is distributed training, accelerating the training of our autonomous system by harnessing computing resources for parallel processing. This not only expedites the learning process but also facilitates the collection of diverse environments, enhancing the robustness of training. We present two primary distributed training paradigms: the *centralized distributed experience collection paradigm*, inspired by IMPALA [1]; and the *decentralized distributed parameter update paradigm*, inspired by the federated learning approach. This work offers a comprehensive exploration of the detailed methodology, experimental results, and performance analysis for both distributed paradigms tested across various RL environments. Our findings mark a significant stride toward developing autonomous systems with heightened efficiency and precision, adept at navigating complex environments. The codebase of our implementation for the distributed RL research thrust is publicly accessible via GitHub [1].

1 Introduction

The advancement of autonomous vehicles navigating intricate road conditions owes much to the advent of powerful computer vision models which enable efficient and effective pattern recognition and visual understanding. These vehicles leverage an array of sensors to collect real-time data, analyzing and learning from it using machine learning algorithms [2]. Despite these accomplishments, traditional reinforcement learning and large-scale computer vision architectures pose challenges in real-time applications like racing cars. Their reliance on extensive data and extended training and inference times can compromise reliability.

In response to these challenges, Learn-to-Race (L2R) [3] emerges as a multimodal environment within the Arrival Autonomous Racing Simulator. Engineered to deliver robust and efficient autonomous racing solutions, L2R empowers agents through sensory inputs, computer vision encoders, and distributed reinforcement learning agents. However, prior approaches and iterations of the project predominantly focused on the sequential training of RL agents, which greatly limited the scalability when multiple GPUs or computing resources were available. In particular, a traditional sequential reinforcement training paradigm that performs agent update, environment simulation, and experience collection and sampling, is limited. We aim to utilize distributed training to support the paralleled experience collection from diverse environment settings, and fully utilized multiple GPUs to perform

[1] https://github.com/BrandonBian/l2r-distributed

parameter updated, as illustrated in Fugre 1. Acknowledging the pivotal role of powerful GPUs in accelerating training processes, we believe the design of distributed systems and training paradigms that harness their capabilities for parallel processing can be particularly useful for the scope of the L2R project.

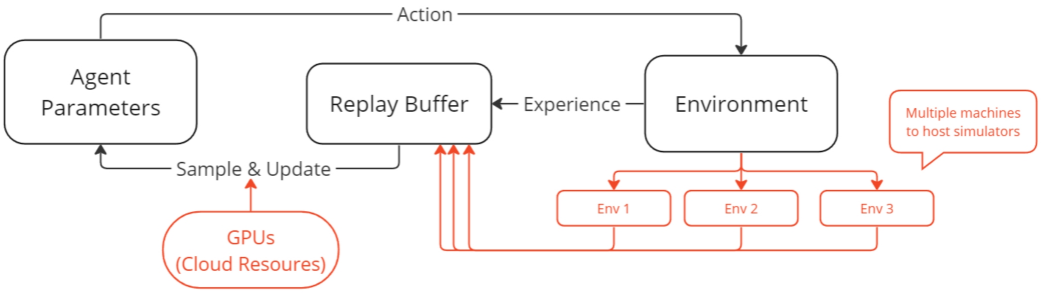

Figure 1: Motivation of using distributed training to support diverse environment simulation and utilize GPU resources

2 Hypothesis/Project Goal

1. **Centralized Learning Bottleneck**: We hypothesize that the observed degradation in the performance of the current distributed learning setup for RL agents from the previous iteration stems from the rapid generation of simulation data by the worker. This rapid influx of data poses a challenge for the centralized learner, turning it into a bottleneck that struggles to effectively handle the data.
2. **Distributed Training Scalability**: We hypothesize that distributing training tasks across multiple actors has the potential to alleviate the burden on the learner, consequently reducing training time. This distributed approach aims to enhance the scalability of the RL training process, allowing for the management of larger and more complex environments. Additionally, the utilization of distributed training is expected to tap into the additional computational resources available in a cluster or cloud computing environment, resulting in accelerated training times and improved performance compared to a non-distributed architecture.
3. **Importance Sampling and Replay Ratio Tuning**: We hypothesize that implementing importance sampling and replay ratio tuning methods will enable the identification of useful feedback while discarding irrelevant information. This targeted approach aims to enhance the quality of the experience data collected in a distributed manner, thereby contributing to improved training outcomes.increasing the interpretability and effectiveness of the distributed training process.
4. **Modularization of Paradigms and Environments**: We hypothesize that the modularization of paradigms and environments within a unified framework is anticipated to provide several benefits. It offers flexibility for experimenting with different combinations of paradigms and RL environments. This modularity also facilitates efficient debugging, allowing the testing of various paradigm and environment combinations. Moreover, the modular approach benefits future iterations of development by enabling the seamless integration and removal of different environments and paradigms for iterative testing and improvement.

3 Relationship to Prior Work

The standard IMPALA [1] architecture has been utilized in the previous work, where one centralized learner updates the RL agent's parameters, and multiple workers generate reward and replay buffers from performing simulations. More specifically, the centralized learner broadcasts its parameters to

the workers so that the workers can periodically synchronize and reproduce the learner's agent state; the workers perform simulation on their local copy of environment, and send the collected experience back to the learner for centralized agent update, as illustrated in Figure 2. However, previous iterations of distributed RL faced a significant issue where the reward did not converge to a maximum effectively, even with an increasing number of workers. This degradation of performance is a primary problem as the distributed methodology should provide faster training and convergence speed due to more resources running in parallel and should at least perform similarly to the sequential version. In the current iteration of the project, we aim to address this issue by profiling the performance and workload distribution among the workers and learner, as well as the update speed, and hypothesize that the degradation of performance may be due to the overhead on the learner when the worker generates buffers too fast, hyperparameter tuning, and network latency. To resolve this, we will focus on two aspects: (1) moving the training to the workers so that the learner does not become a bottleneck and (2) implementing importance sampling with (1) to filter only the most useful replay buffers, reduce contention and confusion, and mitigate the issues that arise with an excessive number of buffers generated by the workers.

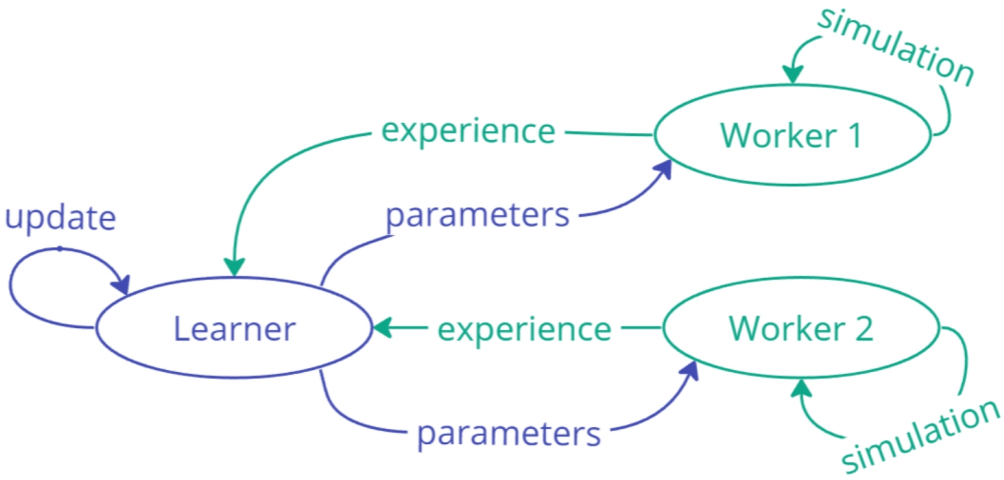

Figure 2: The distributed architecture from prior iteration, where the learner is responsible for agent update, and the workers are responsible for environment simulation and experience collection

In addressing the first aspect, we aim to delve into a distributed training paradigm similar to the federated learning process. A recent research paper on distributed RL paradigms [4] provided many useful insights. Specifically, prior works derived from the IMPALA architecture, such as GORILA [5], have sought to overcome challenges similar to those outlined earlier. GORILA achieves this by decoupling parameter synchronization and experience collection into two distinct functional modules that feature a parameter server and a dedicated learner. Similarly, the Asynchronous Advantage Actor-Critic (A3C) framework [6] introduces parameter sharing between workers and a global network, facilitating decentralized gradient update tasks. Building upon insights from these established methodologies, we propose a novel training paradigm. In our design, the learner serves a dual role as a server for worker task allocation and control while functioning as a learner responsible for parameter update and synchronization, thus referred to as the learner server. Distinctly, workers adopt versatile roles as experience collectors, parameter update trainers, or evaluators based on tasks assigned by the learner server. This approach effectively redistributes the training burden from the learner to the workers, eliminating the need for an additional stand-alone server node solely for parameter-sharing control. Furthermore, this design empowers workers to multi-function based on assigned tasks, thereby maximizing the utilization of decentralized computing resources distributed across the worker nodes.

4 Fall Semester Development Goals

The objectives for the fall semester encompass four primary facets, as illustrated in Figure 3. Firstly, we leverage the compatibility of our system with OpenAI's Gymnasium environments to incorporate a medium-difficulty typical 2D environment, named Lunar Lander for additional testing of the distributed paradigm. Secondly, we reproduce the SAC actor-critic algorithm using OpenAI Spinning-up vanilla implementation along with hyperparameter configurations shared by HuggingFace to fine-tune RL agents. After the replication process, a thorough analysis of the obtained results will be undertaken to discern insights and trends that can inform future developments and optimizations. Thirdly, we design and develop a new distributed training paradigm, where the training tasks are allocated to workers to alleviate the workload on the learner. Finally, we aim to systematically restructure the existing codebase, implement automation in resource management processes, and consolidate branches and stand-alone module implementations. This holistic reorganization is poised to untangle the complex interplay between deep reinforcement learning and distributed system design, providing a streamlined and more accessible framework for researchers. The intention is to simplify the replication of baselines and the execution of experiments.

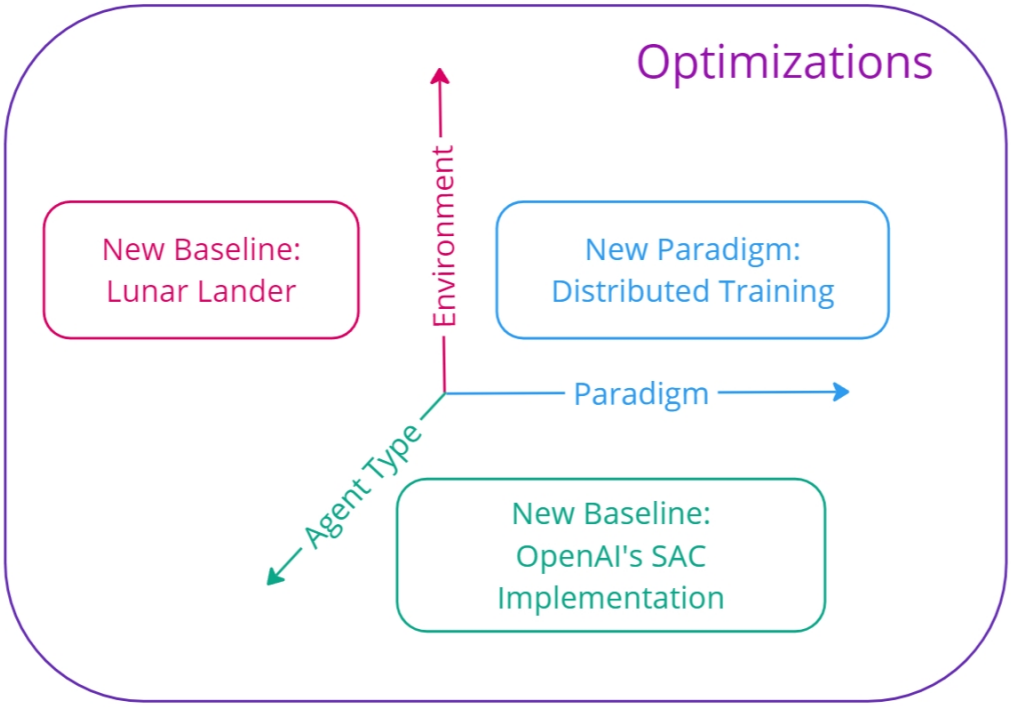

Figure 3: The distributed reinforcement learning and optimization thrust has 4 main dimensions: (1) compatibility with diverse RL environments, (2) implementation of different agent algorithms, (3) new distributed training system design paradigm, (4) optimization of codebase and resource control

5 Project Requirements

5.1 Intended Users

1. **Researchers in academia:** Our system serves as a robust platform for researchers and academics in the autonomous vehicle domain. It offers an environment for the evaluation and refinement of algorithms and sensor systems, thereby enriching the understanding of autonomous vehicle technology. Furthermore, with our modularized code base that allows for the easy plug-in testing of diverse RL environments and training paradigms that are fully

compatible with the L2R and OpenAI Gymanisium environments, researchers can easily explore innovative approaches to enhance the performance of their autonomous driving models.

2. **Businesses:** For businesses focused on self-driving car development, our system provides a valuable resource for optimizing vehicle performance. It offers insights derived from extensive experiments and ablations, eliminating the need for costly real-world testing. Through thorough analysis, businesses can pinpoint areas for improvement and fine-tune their systems, resulting in more efficient and reliable transportation solutions.

5.2 Functional Requirements

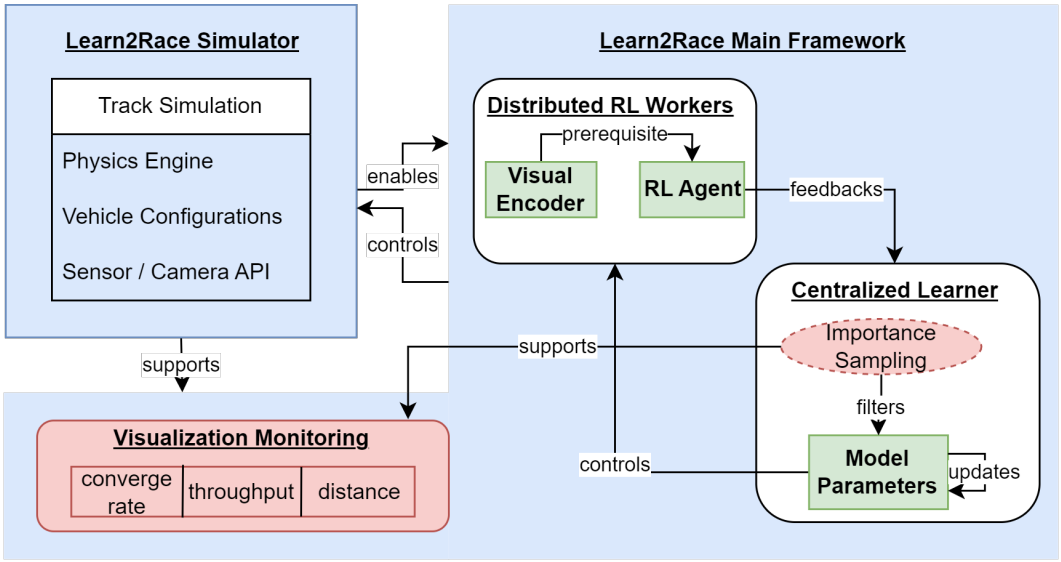

Figure 4: Context diagram for the distributed RL research thrust of Learn-to-Race

Figure 4 offers a comprehensive overview of the context diagram central to the proposed distributed RL and optimization research thrust. Specifically tailored for the Learn-to-Race project, this thrust plays a foundational role, serving as the backbone of the entire system architecture. The simulator captures visual inputs from the simulated environment. These inputs undergo encoding through visual encoders and are subsequently relayed to the RL agent. The RL agent, in turn, is responsible for selecting the most suitable action based on the interpreted status of the current environment. The distributed paradigm takes charge of automating and orchestrating communication between the learner server and the workers. Within decentralized workers, all crucial steps—simulation, experience collection and evaluation, as well as RL agent steps and updates—are carried out. The learner, on the other hand, handles the synchronization and updating of parameters collected from all distributed workers. The resulting action learned by the RL agent is then transmitted back to the environments within the workers. This action triggers the subsequent iteration of the simulator, generating visual input for the next timestamp.

1. The users running the distributed RL training architecture should be able to observe substantial performance improvement with the addition of distributed workers and computing resources. The current distributed framework observed a degradation of performance in terms of learning convergence. This is why we should distribute the training workload to workers to improve convergence speed.
2. We aim to provide users with a modularized code space for plug-in testing of diverse training paradigms and RL environments. Furthermore, the workspace should be compatible with OpenAI's Gymnasium library of common RL environment collections. The

current distributed framework lacks modularization and unification of code, where different combinations of training paradigms and RL environments are segregated into distinct branches, controlled by standalone Kubernetes configuration files. This fragmented structure poses challenges, particularly in implementing systematic changes uniformly across all RL environment and training paradigm designs.

5.3 Non-Functional Requirements

One of the primary nonfunctional performance goals is to mitigate any hindrance in terms of both speed and convergence during the training of autonomous racing agents. Users should seamlessly transition to the distributed framework without significant differences or latency issues. In terms of usability, users should be able to effortlessly set up the system and run existing implementations, reproducing baselines within a short timeframe. The unified implementation across paradigms and environments, coupled with a modularized coding style, ensures compatibility with the software and hardware requirements of the existing framework. Our goal is to guarantee a smooth experience for intended users, particularly under the distributed RL thrust, where the system intertwines with deep reinforcement learning algorithms.

5.4 Resource Requirements

To conduct our experiments and train models, we leverage the data generated by the Arrival Simulator, which is accessible on the Learn-to-Race challenge website[3]. This simulator produces datasets from various tracks featuring diverse backgrounds and environments, including *Track01: Thruxton*, *Track02: Anglesey*, and *Track03: Vegas*. Given the implementation of distributed training paradigms, ample GPU resources are essential. Our primary resource for this project is the Carnegie Mellon University (CMU) Phoebe cluster, equipped with batches of GPUs such as the 2080 Ti, enabling parallel testing of diverse training paradigms. Additionally, tools like Weights-and-biases, Docker, and Kubernetes are utilized for experiment logging, environment encapsulation, and resource control, respectively. Emphasizing automation of configurations and resource management steps, we aim to minimize hardcoding, ensuring scalability and ease of use, as outlined in the non-functional requirements section.

6 Experiment/System Design Overview

Illustrated in Figure 5 is a high-level overview of the system architecture for the distributed RL thrust (more detailed implementation configurations and diagrams are included in the following sections). The following three components (corresponding to the three blue blocks in the figure) are the main building blocks for the distributed architecture, and all training paradigms within the following methodology sections are built as the variations of these modules:

6.1 Learn-to-race Simulator

As highlighted earlier, our experimentation relies on the Arrival autonomous simulator, equipped with sensors like cameras and auxiliary devices such as an inertial measurement unit. This simulator creates a controlled racing environment, configured with a variety of maps and tracks, ensuring a wealth of training samples for the RL agent. The simulator captures raw visual data representing the current state of the racing agent through interfaces like cameras, sensors, and actions. These data are then transmitted to the distributed RL workers, where they are used to generate the representation (embedding) of the current environmental state.

6.2 Distributed RL Workers

Upon receiving the raw simulator output, the RL workers employ visual encoders to produce embeddings, serving as latent representations of the current environment state. These representations are then forwarded to the RL agent (in our baseline scenario, the SAC agent), which selects the

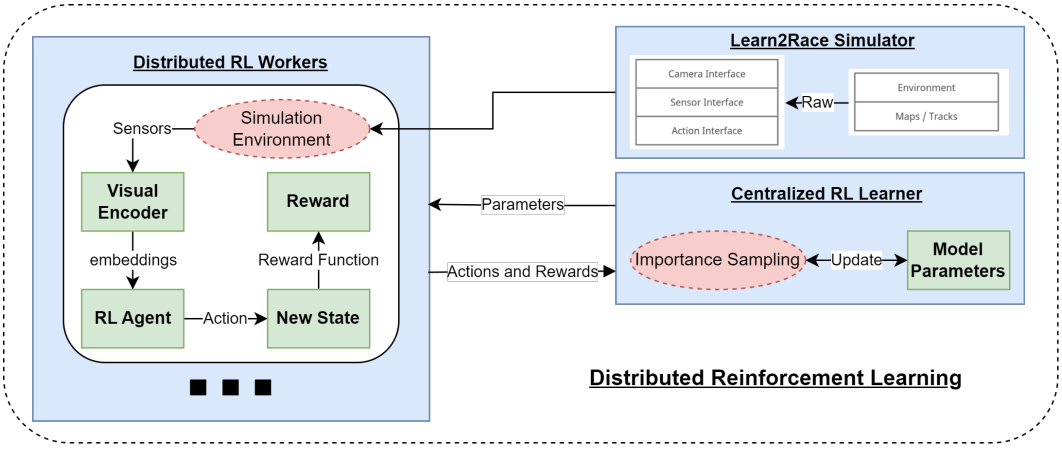

Figure 5: High-level Overview of System Architecture for the Distributed RL Thrust

most optimal action based on its interpretation of the current state representation. Following the execution of an action, such as adjusting the current direction or angle, the new state is leveraged to compute a reward, providing an assessment of the racing car's performance. Subsequently, the action taken and the corresponding reward are encapsulated as an experience instance and transmitted to the centralized RL learner module.

6.3 Centralized RL Learner

Ultimately, the centralized RL learner initiates parameter updates after collecting actions and rewards from the distributed RL workers. It disseminates the latest policies to these workers, encompassing the updated agent parameters and the pertinent training or inference settings. It's essential to highlight that in the latter version of the training paradigm, the centralized RL learner assumes the role of task allocation and assignment to the workers, while the workers shoulder the added responsibility of implementing model parameter updates.

7 Experimental Design

7.1 Dataset

We will be using the Arrival Autonomous Racing Simulator to generate the simulation environment for our reinforcement learning agent. The simulator provides a realistic racing environment that simulates the physics and mechanics of a real-world car race. In our distributed learning scenario, we will perform a simulation in the Arrival Autonomous Racing Simulator's environment by letting the car take an action, then run all the way until it cannot progress any further, and then collect the feedback as the replay buffer for further update of the agent. This process will be performed by each of the workers in parallel, and the collected replay buffers will be sent to the learner for parameter updates.

7.2 Machine Learning Models/Algorithms/Pipeline

7.2.1 Agents and Environments

We chose to use the Soft-Actor-Critic (SAC) [7] algorithm as our base reinforcement learning model due to its effectiveness in continuous control tasks and its ability to handle stochastic policies. SAC is a model-free, off-policy algorithm that optimizes a stochastic policy with entropy regularization. The variational autoencoder (VAE) [8] is used as a preprocessor to learn a compact representation of the high-dimensional image inputs from the racing simulator. This helps to reduce the dimensionality

of the input data, making the learning process more efficient. Furthermore, we intend to replace the existing VAE with the encoders from the visual representation learning thrust to combine the development of both thrusts in one.

In addition to the SAC algorithm, we also used the Bipedal Walker and Mountain Car agents from OpenAI Gym [9] as the baseline for distributed algorithm development. These agents are simple and well-established models that are commonly used in reinforcement learning research. By using these agents as baselines, we can more easily compare the performance of our distributed architecture and analyze any potential issues.

7.2.2 Centralized Distributed Experience Collection Paradigm

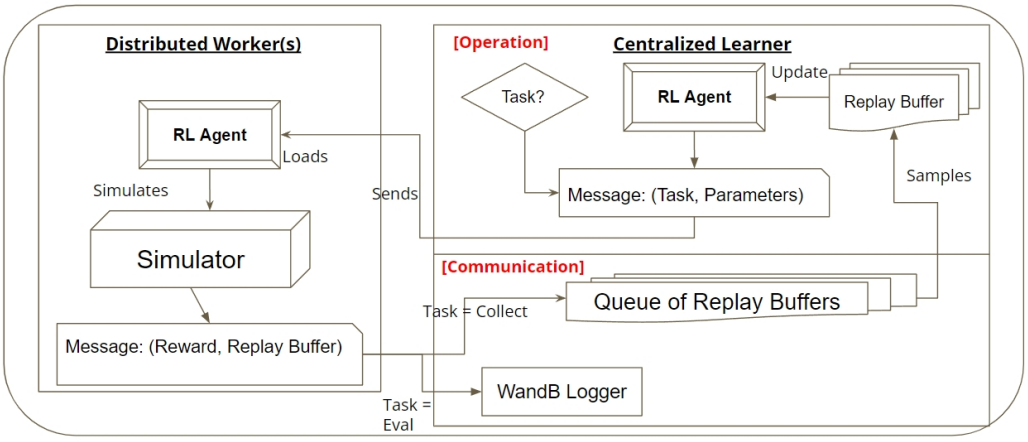

Figure 6: System pipeline for Centralized Distributed Experience Collection Paradigm

The first training paradigm, referred to as the Centralized Distributed Experience Collection Paradigm and depicted in Figure 6, centers on the role of the centralized learner in updating the RL agent. Periodically, the learner samples a batch of experiences from the replay buffer, which originates from distributed workers. The RL agent outputs its current parameters and a binary task, indicating whether the worker should engage in experience collection via simulated steps or perform evaluation greedily. These details are encapsulated in a message and broadcasted to the distributed workers. It's crucial to note that, given the presence of multiple workers, the learner can allocate diverse tasks to different workers, such as some workers focusing on evaluation while others engage in experience collection. However, the shared parameters must remain consistent across all workers to ensure synchronization.

Upon receiving parameters from the centralized learner, the distributed workers load them into their individual RL agent copies, overwriting any prior parameters to replicate an identical RL agent state. Depending on the allocated task, a worker will either collect experience by probabilistically exploring the action space or conduct evaluation by taking the optimal greedy action and recording the corresponding reward. Finally, the worker encapsulates the reward and collected experience within a replay buffer, sending this information back to the centralized learner.

It's worth noting that the centralized learner concurrently runs two sub-processes: the *operation* process, responsible for updating the RL agent and broadcasting messages, and the *communication* process, continuously listening to feedback messages from the workers. In this paradigm, the communication process aggregates all messages from all workers, placing them into a large queue of replay buffers, which the operation process periodically fetches and samples.

In summary, for this distributed training paradigm, the learner's responsibility is three-fold: (1) updating the RL agent by continually sampling batches of experience from the replay buffer; (2)

allocating tasks and broadcasting the tasks along with its latest parameters to the distributed workers; (3) collecting and storing the responses from the workers. In this case, the bottleneck exists for (1), where the replay buffer can be overflowed due to an abundance of workers working collaboratively to generate experience, whereas the learner is overwhelmed due to the slower process of updating the parameters.

7.2.3 Decentralized Distributed Parameter Update Paradigm

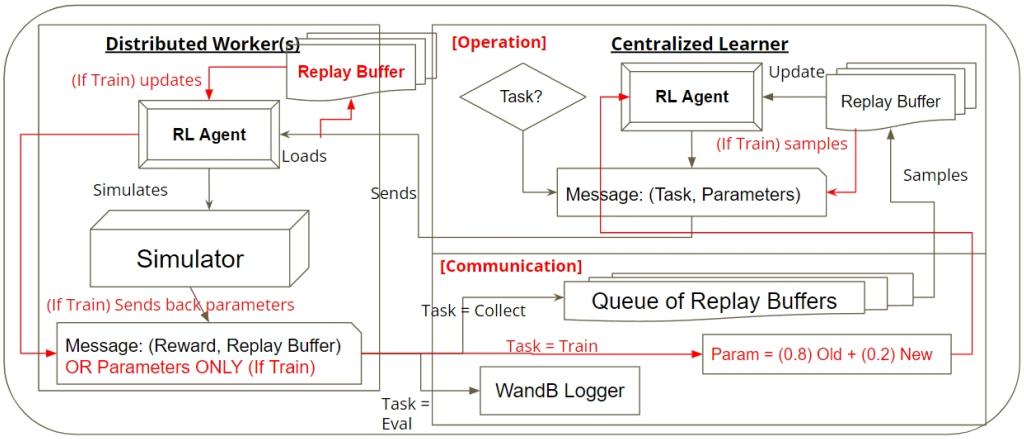

Figure 7: System pipeline for Decentralized Distributed Parameter Update Paradigm

For the second training paradigm, we introduce modifications to the first paradigm aimed at alleviating the parameter update workload from the centralized server. Two significant changes are implemented: first, the introduction of a new task type called "training," assigning the responsibility of parameter updates to distributed workers; second, the addition of replay buffers to these workers, allowing them to locally accumulate collected experiences for subsequent sampling and updating. As depicted in Figure 7, the highlighted modules represent the newly integrated components. When the task assigned is training, a distributed worker updates its local RL agent by sampling a batch of experience from its local replay buffer. In this process, it skips the simulation step since it doesn't interact with the environment during parameter updates. The worker then encapsulates the updated parameters into a message, sending it to the learner. The learner, in turn, performs a weighted update of its own RL agent. Notably, this update is less intricate than a standard update involving gradient calculations—it simply entails overwriting parameters. In adopting this approach, we effectively distribute the sampling and updating procedure from the learner to the workers, allowing the workers to assume this responsibility and mitigating the bottleneck on the learner. In this paradigm, the learner's role is akin to that of a server, primarily tasked with assigning and coordinating responsibilities among workers and serving as a container for the most up-to-date RL parameters. These parameters are periodically broadcasted to the workers for RL agent synchronization through the evaluation task.

7.3 Evaluation Metrics

Since the distributed RL thrust focuses most on the effectiveness of the system design level of the project, the performance of the RL agent both in terms of convergence speed and convergence quality, is considered as the primary evaluation metric. The performance of the VAE is not considered a metric within our scope. The relevant metrics include:

1. Total Distance: The total distance that the car has traveled during an episode.
2. Total Time: The total time taken to complete the race by the car during an episode.

3. Number of Infractions: The number of infractions committed by the car during an episode, such as collisions with track barriers or other cars.
4. Average Speed: The average speed of the car during an episode, measured in kilometers per hour (kph).
5. Average Displacement Error: The average distance between the car's position and the ideal racing line during an episode.
6. Trajectory Efficiency: A measure of how well the car is able to follow the racing line during an episode, expressed as a percentage.
7. Trajectory Admissibility: A measure of how closely the car is able to follow the racing line during an episode, expressed as a percentage.
8. Movement Smoothness: A measure of how smoothly the car moves during an episode.
9. Timestep/sec: The number of timesteps per second during an episode.
10. Laps Completed: The number of laps completed by the car during an episode.

8 Test Design and Model Deployment

8.1 Experiment Launching

As emphasized earlier, the automation of resource control is critical for complex distributed training paradigms, facilitating seamless baseline reproduction without entangling users in intricate Kubernetes YAML file configurations. As depicted in Figure 8, users can initiate a Python script, input their RL environment and training paradigm, specify the number of workers for distributed paradigms, and provide a name for the Weights and Biases experiment. Subsequently, the script dynamically generates a Kubernetes YAML file based on the user's input configuration and launches the setup. Notably, a key optimization is the elimination of static and hard-coded YAML file configurations from the previous iteration, replaced by the on-the-fly generation of YAML files. This not only significantly reduces the number of required YAML files but also simplifies the process of propagating configuration changes.

```
sbian@phoebe-login:~/distributed$ python3 launch.py
Select RL environment (mcar/walker/l2r): mcar
Select training paradigm (sequential/dCollect/dUpdate): dUpdate
Input number of distributed workers: 1
Input WandB experiment name: mcar-dupdate-1worker-test
----------
RL Environment = [mcar] | Training Paradigm = [dUpdate] | Number of Workers = [1] | Experiment Name = [mcar-dupdate-1worker-test] ---
----------
replicaset.apps/mcar-dupdate-workers created
pod/mcar-dupdate-learner created
service/mcar-dupdate-learner created
```

Figure 8: Automated launching process for model deployment and experiment initiation

8.2 Experiment Termination

Manual termination of distributed resources poses challenges, as users must locate specific pods, services, and replicasets, and execute Kubernetes commands for termination. A simple typo in these commands can inadvertently lead to the deletion of an unintended pod, risking the loss of crucial training progress. In response to this challenge, we've developed an automated script, showcased in Figure 9, which gracefully shuts down and terminates all pertinent resources based on the user's specified configurations. This automated approach minimizes the risk of inadvertent errors and enhances the efficiency of resource shutdown procedures.

9 Risks/Challenges

The risks and challenges of the proposed thrust for the learn-to-race project is three-fold:

```
sbian@phoebe-login:~/distributed$ python3 shutdown.py
Select RL environment (mcar/walker/l2r): mcar
Select training paradigm (sequential/dCollect/dUpdate): dUpdate
---
--- Shutting down [mcar] with training paradigm [dUpdate] ---
---
replicaset.apps "mcar-dupdate-workers" deleted
service "mcar-dupdate-learner" deleted
pod "mcar-dupdate-learner" deleted
```

Figure 9: Automated termination process for model deployment and experiment initiation

9.1 Business Risks

During the consideration of product design aspects, a notable business risk has been identified. The product is targeted for deployment in competitive environments where businesses or individuals rely heavily on its performance. Any shortcomings in the product's performance, whether in a simulated or real-world racing scenario, could have adverse effects on the business or user experience. Potential issues, such as slow responsiveness or the inability to deliver real-time inputs, especially in challenging weather conditions, may lead to a decrease in the user's fanbase and a loss of interest from sponsors. This, in turn, could result in negative financial repercussions for the business or individual involved. Addressing and mitigating these performance-related risks is critical to ensuring the product's success and maintaining stakeholder satisfaction.

9.2 Product Risks

We've pinpointed a critical product risk that could lead to failure when the product is deployed in a real car participating in a competition, whether with or without a human driver. In the event of an incorrect prediction during a race that leads to a crash, there is a potential for loss of life or damage to infrastructure, depending on the circumstances. Such an incident would deem the product unsafe for real-world use and could result in substantial financial losses for the business. Consequently, it is of utmost importance to proactively prevent such situations before deploying the product in real-life scenarios, emphasizing thorough testing and validation procedures.

9.3 Project Risks

The project's scope is inherently limited by constraints such as the available data, short project duration, and budget considerations. Collecting an exhaustive set of data to enable a machine learning model to adapt to every conceivable use case is not feasible within these limitations. As a consequence, there might be instances in both real-world and simulated environments where the model struggles to comprehend and adapt to unfamiliar conditions. For instance, if the model is exclusively trained in sunny environments and encounters sudden foggy conditions, its performance may deviate from intended outcomes, potentially yielding inaccurate or only partially accurate results. This lack of adaptability introduces a challenge, potentially rendering the product less reliable and robust, particularly in diverse environmental conditions.

10 Tools and Dependencies

10.1 Dataset

1. The Learn-to-Race framework utilizes simulations of three real-world racetracks, adding a layer of authenticity to the racing experience:

(a) North Road track at Las Vegas Motor Speedway in the United States.
(b) Thruxton Circuit track located in the United Kingdom.
(c) Anglesey National, modeled after the track at the Anglesey Circuit.

2. Each dataset comprises images captured by the vehicle's camera.
3. For instance, the Thruxton dataset encompasses 10,600 complete transitions, including sensor data, camera images, and actions executed by the Model Predictive Control (MPC) agent. This data was gathered over 9 full laps around the Thruxton circuit track.
4. All images are presented in RGB format, with dimensions of 512 pixels in width, 384 pixels in height, and a field of view spanning 90 degrees.

10.2 Libraries/Services

1. Our Learn-To-Race framework will be developed using Python 3.8, adhering to current standards.
2. The framework will leverage key Python libraries:
   (a) PyTorch version 1.7.1 for the creation of our deep learning models.
   (b) Torchvision for image transformation in computer vision tasks.
   (c) Numpy version greater than 1.19.2 for various computations.
   (d) Matplotlib version greater than 3.3.2 for visualizing the racetrack map.
3. The Learn-To-Race framework mandates GPU support to facilitate rapid visual feature extraction and the training of our safe Reinforcement Learning algorithms.
4. Docker is a prerequisite for the framework, as the simulator typically operates within a Docker container.
5. Additionally, running the simulator within a Docker container necessitates the use of Nvidia-container-runtime to access the GPU from the container.

11 Progress/Results

11.1 Training Paradigm Implementation

As a recap of the past semester, we conducted a comprehensive performance analysis of the original distributed learning architecture, focusing on the L2R, bipedal walker, and mountain car RL agents. As outlined in the preceding sections, a significant challenge emerged in the form of deteriorating training performance, even with an increased number of distributed workers. Notably, there was no substantial acceleration in convergence, and in some instances, models failed to converge even after an extended duration, despite scaling up the number of workers engaged in parallel data collection and model evaluation. The suspected root cause of these issues pointed to the centralized learner acting as a bottleneck, given that the model parameter updates were taking longer than the evaluation and collection processes. To address this concern, we introduced the refined *decentralized distributed parameter update paradigm*, a modification of the original system, as illustrated in Figure 7.

Several major refinements have been implemented in our system to enhance its functionality: (1) Introducing a novel task type called *Train*; when the learner assigns this task to coordinate the workers, it includes a message containing a list of replay experiences sampled from its own replay buffer; (2) Upon receiving a *Train* task, a worker loads the parameters, performs parameter update (training) based on the provided list of experiences, and generates a response message with the trained parameters; (3) After training, the worker directly sends the response message back to the learner, bypassing any simulation steps; (4) The learner executes a damped update of its parameters using the formula $P_{n+1} = 0.8 * P_n + 0.2 * P_{new}$, where n denotes the $n - th$ update iteration, and P_{new} represents the newly trained parameters obtained from the worker.

11.2 Code Base Unification

As highlighted earlier, a key aspect of our system is to furnish users with a unified and modularized codebase. This entails consolidating similar functionalities into shared modules or classes, fostering a cohesive and adaptable structure. Illustrated in Figure 10, five main steps were undertaken to achieve this unification. In the initial step, we streamlined the number of branches. In the preceding iteration, various combinations of RL environments and training paradigms were housed in separate branches, complicating the propagation of framework changes. By consolidating these branches into two—dedicated to distributed and sequential paradigms—we merged them seamlessly. This unification allows users to access and modify all code within a singular branch. The second step focused on unifying Kubernetes launching YAML files. Previously, each configuration had a dedicated YAML file, a complexity mitigated by our automatic resource creation template generation, reducing myriad combinations to two basic templates. Steps three and four addressed the unification of the worker runner class. Previously, distinct classes for each training paradigm led to redundancy and intricate modifications. We amalgamated these classes into one, capable of inferring the training paradigm based on input, simplifying worker configuration YAML unification. Lastly, we homogenized learner and worker definitions, eradicating any redundancies.

Figure 10: Procedure on unifying distributed training paradigms within learn-to-race RL environment space

11.3 Automated Resource Control

Another significant advancement involves the implementation of automated resource control, as detailed in Section 8. In the previous iteration, hardcoded pod and service names led to port conflicts when launching multiple configurations simultaneously, impeding communication between workers and their respective learners. In our revised iteration, we automated both resource creation and

termination processes. Additionally, we dynamically assign port and resource names to services based on the user's experiment name input. This not only simplifies resource tracking but also resolves conflicts, enabling concurrent testing of multiple RL environments and training paradigm combinations.

11.4 Experimental Results

To assess our two distributed training paradigms, we conducted a total of nine experiments, encompassing various combinations of RL environments {learn-to-race, mountain car, bipedal walker, bipedal walker with OpenAI's vanilla SAC agent, and lunar lander with OpenAI's vanilla SAC agent} and training paradigms {sequential, distributed collection, distributed update}, with the results summarized in Figure 11. In the subsequent sections, I will present the latest results for each of the RL environments, along with high-level observations and analysis.

- Legend: ✓ means convergence, ~ means non-convergence

RL Environments	Sequential	Distributed Collection	Distributed Update
Mountain Car	✓ (slow, ~100 h)	✓ (faster with more workers)	✓ (faster with less workers)
Bipedal Walker	~ (reward < 0)	~ (periodical surges)	~ (can cross 0, fluctuations)
Bipedal Walker (OpenAI)	~ (rough converge to reward near 0)	~ (no convergence)	~ (no convergence)
Lunar Lander (OpenAI)	✓ (~10 h, no sign of degradation)	✓ (reach high reward <2h, with *slight degradation*)	~ (reach high reward <1h, but then degradation)
Learn-to-Race	✓ (speed KPH)	~ (stay at low performance)	~ (periodic improvement)

Figure 11: The combinations of RL environments and training paradigms, along with summarized experimental results

11.4.1 Mountain Car Results

We experimented with the updated architecture on the **Mountain Car** RL agent first, since it is the most fundamental RL agent that can serve as a sanity check. As recorded in Figure 12, we observe a significant increase in convergence speed with a scaling of the number of workers for the Mountain Car RL agent (where the sequential version takes around 100 hours to converge, while both distributed paradigms achieved convergence within 40 hours), which indicates that the updated architecture passed the sanity check and can converge with boosting from more paralleled training. Furthermore, we observed an interesting phenomenon where the distributed update paradigm works better with less number of workers, while the distributed collection paradigm converges fastest with more workers. We hypothesis that this is because the mountain car environment is easy to solve, so it does not benefit much from the distributed parameter update paradigm, which may induce further network communication overheads. In this case, a distributed experience collection paradigm that retains centralized parameter update scales the best.

11.4.2 Bipedal Walker Results

We also conducted experiments on the bipedal walker RL environment, which is a much more difficult environment, and compared the two training paradigms, as depicted in Figure 13. We observe that there is no significant convergence on a global scale for both paradigms. However, for the centralized

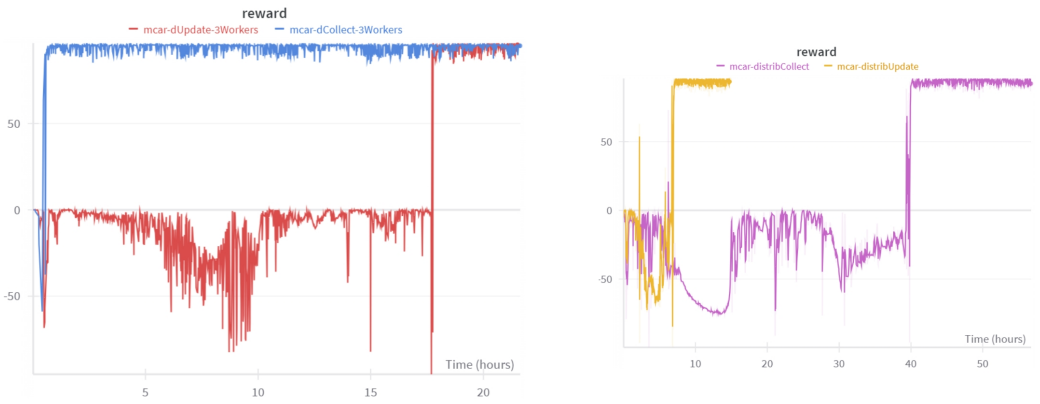

Figure 12: Experimental results for the **Mountain Car** environment, solved by the distributed paradigms with 3 workers (left) and 1 worker (right)

distributed collection paradigm, there are periodic surges of rewards, indicating that the agent is periodically learning to stand up but experiences forgetting of their experience and degradation of performance. In comparison, for the decentralized update paradigm, there is more fluctuation in the reward, and we see temporary convergence to a non-negative value before the reward drops once again, indicating that the agent can learn successfully at one point but suffers from a similar forgetting as before.

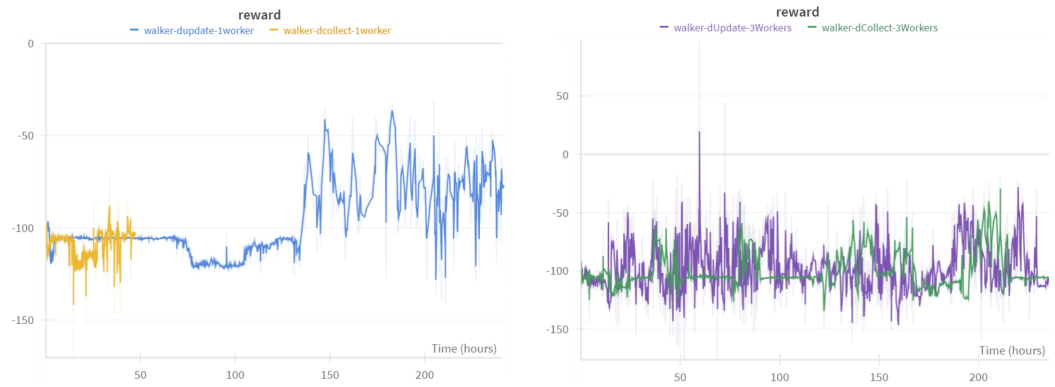

Figure 13: Experimental results for the **Biepdal Walker** environment, solved by the distributed paradigms with 3 workers (right) and 1 worker (left)

11.4.3 Lunar Lander Results

In our effort to test the distributed paradigms with more comprehensive environments, we conducted experiments on the Lunar Lander RL environment with OpenAI spinning-up vanilla implementation of the SAC agents, with the results recorded in Figure 14. We observed that both distributed training paradigms can achieve high rewards within a very short period, but suffer from degradation forgetting, something that is also observed in the bipedal walker case. From these results, we hypothesis that the distributed training paradigms, having the ability to collect diverse experiences from multiple environments in parallel, has the potential to achieve higher reward in a shorter amount of time than sequential paradigms, yet suffers from forgetting possibly due to the more frequent reset and update of the parameters, thus cannot maintain the high rewards parameter states.

Figure 14: Experimental results for the **Lunar Lander** environment, solved by the distributed paradigms with 3 workers

11.4.4 Learn-to-race (L2R) Results

Finally, we present a comparison of the learn-to-race environment agent convergence performance between the two training paradigms in Figure 15. We focus on two main metrics: *Average Speed (KPH)*, representing the average speed of the car during a simulation episode, and *Reward*, calculated as how well the car maintains on the central line of the race lane (and each time it collides with the edges of the lane, rewards are deducted). The expectation is that both metrics increase as the car/agent learns to navigate the environment effectively. From the figure, a notable observation emerges. The decentralized distributed update paradigm exhibits signs of convergence, evident in the increasing values of both speed and distance. However, this convergence or improvement in metrics happens periodically and drops to a valley before re-emerging to a high value. These periodic fluctuations align well with our observations in the prior environments.

In contrast, the centralized distributed collection paradigm appears to be stuck at a lower performance level. We hypothesize that the superior performance of the distributed update paradigm can be attributed to the inherent difficulty of the learn-to-race environment. In this paradigm, each worker can independently update and train its local copy of the RL agent before merging parameters at the learner. This approach facilitates more diverse and parallelized learning, particularly beneficial for a complex and challenging environment. Further investigation into this observation will be explored in the subsequent error analysis section.

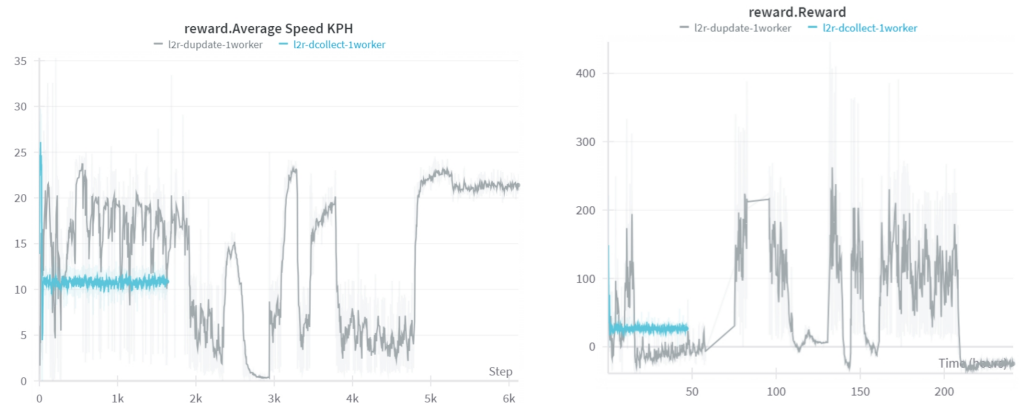

Figure 15: Experimental results for the **Learn-to-Race** environment, solved by the distributed paradigms

12 Error Analysis

From the results discussed in the prior section, we observed that differences across RL environments, training paradigms, and the number of distributed workers significantly influence the training performance. Therefore, we summarized the results in Figure 11. In the mountain car RL environment, which is considered one of the easiest environments available, our analysis suggests that there is no fundamental issue with the training paradigm, as indicated by the observed convergence of the agent within this straightforward environment. Specifically, for the centralized distributed update paradigm, the addition of more workers seems to lead to a scalable increase in convergence speed (Figure 12), but only when compared to experiments run on the same training paradigm. The reason why the decentralized paradigm converged slower than the centralized paradigm when incorporating multiple workers may be explained by the overhead of message and parameter communication between the workers and the learners, which becomes redundant and particularly burdensome when the environment is easily solvable. This leads to the observation and hypothesis that *for easily solvable environments, the centralized distributed paradigm may outperform the decentralized paradigm.*

Moving on to analyzing the results of the bipedal walker, as mentioned above in Figure 13, we observe that the non-convergence issue persists with both paradigms. However, it is worth noting that the decentralized distributed parameter update paradigm, despite leading to more fluctuations and reward noises due to the parallel update of parameters, achieved higher convergence than the centralized distributed experience collection paradigm. We hypothesize that this is due to the alleviation of the workload on the centralized learner, which is significant when the environment is hard to solve. The non-convergence issue, however, may be more due to hyperparameter tuning.

For the performance analysis of the learn-to-race agent, the non-convergence issue observed in the distributed collection paradigm may be attributed to two potential factors. First, the communication overhead could be a contributing factor. Figure 16 demonstrates that when multiple workers are operational, and the data being transmitted is substantial (as is the case with the complex state representation in learn-to-race), the time taken for communication becomes a notable bottleneck. This potentially impedes the timely update of parameters on the learner server. Additionally, the intricate nature of the learn-to-race environment, coupled with the sensitivity of the SAC RL algorithm to initializations, requires careful tuning of a multitude of hyperparameters to mitigate the impact of unfavorable initializations. Further investigation into these aspects is essential to address the observed non-convergence in the distributed collection paradigm.

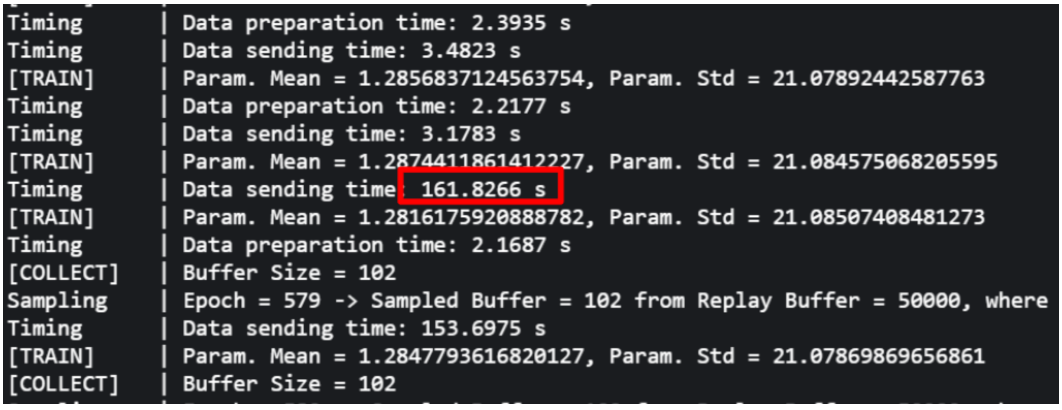

Figure 16: Latency issue due to sending large messages, observed in learn-to-race experiments

13 Discussion

Recorded in Figure 17 are our observations and analytical conclusions derived from the extensive combinations of experiments. Firstly, we observed that for the easier RL environments, such as *mountain car*, the distributed paradigms retain the ability to converge, while successfully accelerating the training speed. We also observe that the distributed update decentralized paradigm suffers from communication and worker coordination overhead, which may mean that for easier environments, falling back to the centralized parameter update is more beneficial due to less randomness and fluctuation of agent model parameters.

Furthermore, for medium and high difficulty RL environments, such as *lunar lander* and *bipedal walker* respectively, we start to observe that the distributed paradigms, despite being able to reach high rewards, cannot retain the high performance without suffering from degradation. We analyze and hypothesize that the faster efficiency in achieving high rewards may be due to the distributed training paradigms' ability to collect experience from diverse environments located within each worker at the same time. So in comparison with the sequential version where the agent performs experience sampling by simulating in a fixed environment, the distributed paradigms allow more opportunity for the agents to explore in different ways, thus giving it a better chance to find an optimal action within a certain state space. This allows the learner to learn diverse and useful parameters, inducing fluctuations but also the potential to reach high rewards. However, due to the frequent and decentralized update of the parameters, it is highly likely that *a newly learned parameter from a certain worker is lost due to being updated by another worker at the same time, causing the performance to degrade as the agent forgets the newly learned knowledge*. To combat this, we believe that a potential way would be to **allow each worker to retain a local buffer for storing all the "good experiences" with high rewards it explored, instead of sending them to the learner and discarding them. Finally, the learner can aggregate the local buffers across all workers, thus obtaining a small pool of highly refined experience for fine-tuning.**

- Legend: ✓ means convergence, ~ means non-convergence

RL Environments	Sequential	Distributed Collection	Distributed Update
Mountain Car	✓ (slow, ~100 h)	✓ (faster with more workers)	✓ (faster with less workers)
Bipedal Walker	~ (reward < 0)	~ (periodical surges)	~ (can cross 0, fluctuations)
Bipedal Walker (OpenAI)	~ (rough converge to reward near 0)	~ (no convergence)	~ (no convergence)
Lunar Lander (OpenAI)	✓ (~10 h, no sign of degradation)	✓ (reach high reward <2h, with *slight degradation*)	~ (reach high reward <1h, but then degradation)
Learn-to-Race	✓ (speed KPH)	~ (stay at low performance)	~ (periodic improvement)

Figure 17: Latency issue due to sending large messages, observed in learn-to-race experiments

14 Lessons Learned and Reflections

A critical reflection on this research thrust underscores the indispensable role played by our earlier documents in establishing a robust foundation for the capstone project and shaping this final report. The *Vision Document* served as a compass, defining the project's scope, goals, and objectives within the distributed thrust of Learn-to-Race. This enabled me to concentrate on tasks tailored to my thrust, unaffected by the progress of other thrusts. The *Requirement Document* proved

instrumental in identifying and prioritizing project requirements, providing a framework for setting goals and planning milestones. The *Design Document* deepened my understanding of modularized dependencies and illuminated potential risks inherent in my design. Lastly, the *Plan Document* facilitated synchronization across research thrusts.

A pivotal lesson learned centers on the paramount importance of communication and timely updates with my supervisor for the success and progress of my research thrust. During a critical phase in development, I encountered challenges in navigating a convoluted and disorganized code base. Seeking assistance from my research assistant mentor proved invaluable, as they aided in reorganizing and modularizing a specific module of my design. This structured approach was then applied to refine all other designs. Going forward, I recognize the necessity of more frequent communication with supervisors and teammates to ensure alignment with requirements, understand progress, and address concerns effectively.

15 Future Work

Integrating distributed systems design and deep reinforcement learning (RL) within the complex domain of autonomous racing presents a multifaceted challenge. As observed in the current study, there is no one-size-fits-all distributed paradigm that universally resolves the intricacies of the learn-to-race environment. Future endeavors in this field should entail a thorough exploration of additional tuning parameters and a deeper investigation into the interactions between distributed system components and RL algorithms. The observed non-convergence issues in certain RL environments, such as the learn-to-race scenario, prompt extensive hyperparameter tuning. The sensitivity of RL algorithms, particularly the SAC algorithm, to initialization parameters underscores the importance of meticulous optimization. Future work should delve into systematic hyperparameter tuning and optimization strategies to enhance the robustness and convergence speed of the distributed training paradigms. The current study focused on four diverse RL environments (learn-to-race, mountain car, lunar lander, and bipedal walker). However, expanding the exploration to a broader array of RL environments will provide a more comprehensive understanding of the strengths and weaknesses of different distributed paradigms. Assessing the scalability and adaptability of the proposed paradigms across a spectrum of environments will contribute to their refinement and generalizability.

16 Conclusion

In conclusion, our efforts to develop robust and efficient solutions for autonomous driving have evolved from a combined perspective of reinforcement learning and system design. The proposed thrust, Distributed Reinforcement Learning with Optimization, aims to enhance the scalability of the reinforcement learning training process by distributing tasks across multiple workers and incorporating real-time visualization for increased transparency. Our innovative approach to training autonomous racing cars using distributed reinforcement learning enables the distribution of workloads and simulations to individual workers, achieving parallel acceleration and fully leveraging scalable computing power.

In this work, we introduced two distributed training paradigms designed to scale the training efficiency of agents within the learn-to-race project environment by delegating tasks to workers and coordinating them with a learner. First, we proposed the *centralized distributed experience collection paradigm*, where workers perform experience collection and RL agent evaluation within their local simulation environments. The learner then collects feedback from the workers and performs a complete update of its RL agent with sampled experiences. Second, we introduced the *decentralized distributed parameter update paradigm*, where workers take on the additional responsibility of updating the parameters for their local RL agents. The learner performs a weighted overwrite of its RL agent based on parameters collected from the workers, thereby reducing the workload on the learner for full updates through batch sampling.

Experimental results on three different RL environments and training paradigms provided valuable insights into the effectiveness of these paradigms and revealed bottlenecks and limitations, prompting considerations for future work in this area. Additionally, the refactoring and cleaning of the code base through unification and modularization enhance the user experience, allowing researchers and users to interact with the system more easily and facilitating the testing of various combinations of RL environments and training paradigms.

17 Acknowledgments

We would like to sincerely thank our mentors, Jonathan Francis and Arav Agrawal, for their continuous feedback and invaluable guidance, steering us in the right direction throughout the project. Additionally, we appreciate Prof. Eric Nyberg for providing a structured outline of the steps to follow. Special thanks to Neeraj Panse for his exceptional support as our TA, patiently addressing our doubts and queries, and contributing significantly to our learning experience throughout the course.

18 Terminology, Definitions, Acronyms, and Abbreviations

1. Reinforcement learning (RL): a machine learning training procedure in which an agent learns how to take actions within an environment based on reward and penalty feedback.
2. Agents: an intelligent artificial entity can perceive its environment and respond reasonably.
3. State: the parameters that define the status of the agent.
4. Action: the action taken by the agent based on its environment and state to maximize the reward function.
5. Environment: the environmental feedback that the agent is within.
6. Reward function: a formulated representation indicating the gain of the agent at a certain state and environment through taking a certain action.

References

[1] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

[2] Joel Janai, Fatma Güney, Aseem Behl, and Andreas Geiger. Computer vision for autonomous vehicles: Problems, datasets and state of the art, 2017.

[3] Kapthpal Sidharth, Agarwal Arav, Qin Yujin, Gangey Tanay, Chian Kevin, Francis Jonathan, Ganju Siddha, Koul Anirudh, Chen Bingqing, Oh Jean, and Nyberg Eric. Learn-to-race: A multimodal control environment for autonomous racing. *CMU MCDS Capstone Project Final Report*, 2022.

[4] Mohammad Reza Samsami and Hossein Alimadad. Distributed deep reinforcement learning: An overview, 2020.

[5] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning, 2015.

[6] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[8] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

A Changes To Previous Deliverables

A pivotal adjustment in our methodology involved the adoption of a new Docker image, specifically, *jingyua*, in terms of the previously utilized *l2r2022* image from the prior iteration. This transition was prompted by the apparent performance improvements observed with the *jingyua* image, particularly in the learn-to-race environment. Initial tests revealed that the learn-to-race agent exhibited a higher initial speed (measured in KPH) when using the updated Docker image. The exact factors contributing to this performance disparity are still under investigation, with our current hypothesis leaning toward additional CUDA support libraries in the *jingyua* image or potential improvements in the simulator version.

Additionally, substantial modifications were implemented in our GitHub repositories. These changes aimed to streamline and enhance the efficiency of our development process. The consolidation of branches, automation of resource creation and termination files, and the unification of modules independent of training paradigms were key strategies employed to simplify the codebase and foster a more cohesive and manageable development environment. These enhancements ensure that future modifications and additions to the codebase can be executed with greater ease and clarity.