Distributed Path Planning

Shahin Jowkar Dris

December 17th, 2024

Abstract

We demonstrate that distributed deep reinforcement learning, specifically using the IMPALA algorithm, can efficiently solve challenging path planning problems. By training on multiple grid-based environments simultaneously and rapidly cycling between them, we achieve faster convergence, improved stability, and better generalization—especially when no single optimal route applies to all tasks. Our revolving mini-batch strategy, which drastically reduces per-environment training intervals, mitigates overfitting and catastrophic forgetting, enabling a single policy to adapt to diverse layouts, changing start/end points, and complex obstacle arrangements.

These results highlight the potential of leveraging distributed RL and scalable architectures like IMPALA to exploit abundant computational resources, whether on a single machine with multiple CPUs or across entire clusters. By dynamically switching between tasks and environments, we unlock robust, adaptive policies suitable for large-scale, real-world path planning challenges.

1 Introduction

Path planning—the problem of finding an effective route between a start and goal location—lies at the core of numerous applications, including autonomous robotics, transportation logistics, and multi-agent coordination. Established classical algorithms such as A* and Dijkstra's method excel in static, well-defined scenarios. However, as operational environments grow more complex, dynamic, and uncertain, these traditional approaches become increasingly insufficient. High-dimensional state spaces, moving obstacles, and rapidly changing objectives require more adaptive solutions.

Deep reinforcement learning (RL) has emerged as a promising strategy for tackling complex path planning tasks by directly learning policies that map states to actions through trial-and-error interactions. Early successes, as seen with Deep Q-Network (DQN), demonstrated that RL agents could handle discrete action spaces in challenging tasks (e.g., Atari games). Subsequent methods like GORILA, A3C, and IMPALA introduced distributed architectures that parallelize training, substantially reducing learning time and improving scalability. These distributed RL frameworks leverage multiple actors, learners, and parameter servers, enabling RL agents to train on vast amounts of experience in parallel.

Despite these advancements, efficiently applying distributed RL to path planning tasks remains nontrivial. Different environments can present drastically distinct optimal paths or state distributions, leading to issues of catastrophic forgetting and suboptimal policies when multiple tasks are trained sequentially. Moreover, scaling beyond single-environment scenarios and adapting to frequently changing start and end points adds complexity.

In this work, we integrate IMPALA—a state-of-the-art distributed RL algorithm known for its scalability and off-policy stability—with Gymnasium-based custom grid environments and RLlib's distributed training framework. We investigate how carefully tuning the per-environment training intervals and rapidly cycling between different tasks (a "revolving mini-batch" approach) can mitigate overfitting and enable a single policy to handle multiple environments, even when they do not share a common optimal solution. By reducing training intervals and rotating tasks frequently, we promote robust generalization across diverse setups, from simple static grids to dynamic ones with evolving start and end points.

Our results demonstrate that distributed RL can effectively tackle path planning challenges at scale. By harnessing the computational resources made available by parallelism and the flexibility of RLlib and IMPALA, we pave the way for more efficient, dynamic, and robust solutions to real-world navigation problems.

2 Related Work

Several studies have applied RL and DRL to path planning tasks, leveraging their ability to handle the complexities of dynamic and high-dimensional environments. For example, research on RL-based path planning in 3D environments has demonstrated success in guiding agents through dynamically changing terrains by utilizing techniques such as Q-learning, approximate policy gradient, and deterministic tree-based approaches. These methods have been evaluated in scenarios with randomly moving obstacles, highlighting trade-offs between computational efficiency and stability, where tree-based approaches, although more stable, incurred higher computational costs compared to other RL methods [**AReinforcementLearningbasedPathPlanningApproach3DEn**

Multi-robot systems have also significantly benefited from reinforcement learning approaches, achieving more efficient group coordination and improved object transportation. For instance, reinforcement learning models such as Double Deep Q-Learning (DDQN) and Proximal Policy Optimization (PPO) have been applied to optimize path planning and motion control tasks in multi-robot settings. These models enable individual robots to navigate to loading positions, as well as control coordinated movement of groups with specific formations to delivery points. Techniques such as semi-supervised learning with algorithms like A* have been used to enhance RL performance by guiding agents during training, effectively combining classical and learning-based methods to ensure collision-free navigation in both simulation and real-world experiments [**acm˙3582133**]. Furthermore, the integration of RL with behavior-based control has improved flexibility in multi-robot systems, reducing the risk of collisions and allowing for dynamic path adjustments in response to environmental changes.

Recent work has also focused on incorporating uncertainty and dynamic constraints into deep RL-based path planning, enabling the development of more robust and adaptive trajectory planning methods. For instance, by leveraging model-free approaches such as Soft Actor-Critic (SAC) and Deep Deterministic Policy Gradient (DDPG), researchers have achieved significant advancements in handling high-dimensional continuous state-action spaces. These methods employ dense reward functions that balance goals such as minimizing the distance to targets and maximizing obstacle avoidance. By using relative rather than absolute positions and velocities in the state space, these approaches enhance learning efficiency and stability, ensuring faster convergence. Moreover, training in simulation environments like PyBullet, which model dynamic and uncertain scenarios with moving obstacles and variable goal positions, has demonstrated high success rates, including zero collisions after 5,000 episodes of training. These advancements highlight the potential for DRL to outperform traditional methods in complex, uncertain environments by adapting to stochastic conditions and ensuring safe, efficient navigation [**frontiers˙883562**].

However, despite these advances, a key challenge remains: scaling up RL-based path planning solutions. As problems become larger, richer, and more dynamic, training times balloon and computational resource requirements rise steeply. Distributed RL offers a natural solution by harnessing multiple machines and devices to collaboratively learn optimal navigation strategies.

3 Background

3.1 Deep Reinforcement Learning

Deep RL combines function approximation, typically via deep neural networks, with the trial-and-error learning paradigm of reinforcement learning. In typical RL algorithms, an agent interacts sequentially with an environment, collecting experience and then based on its rewards from the experience, a gradient update is provided to the update the model's policy parameters. This sequential pattern imposes computational bottlenecks and limits the amount of parallelism that can be exploited [**DistributedDeepRLOverview**].

Deep Q-Learning (DQL) is one such method. In DQL, we extend the traditional Q-learning algorithm in two ways: experience replay and Q-networks. Deep Q-Learning adopts experience replay [**experienceReplay**]; at each time step t, the agent records and saves the experience $e_t = (s_t, a_t, r_t, s_{t+1})$ into a replay memory $\mathcal{D}_\sqcup = e_1, ..., e_t$. Furthermore, to approximate the optimal action-value function (Q-function) $Q^*(s,a)$, it replaces the Q-table with a neural network (Q-network), enabling the model to handle high-dimensional state spaces [**dqlPaper**, **gorila**]. The optimal action-value function (Q-function) $Q^*(s,a)$, is defined as the maximum expected cumulative reward starting from state s, taking action a, and following the optimal policy thereafter:

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

. Training the DQL involves updating the Q-network, parameterized by θ, to minimize the **mean-squared Bellman error** with respect to an older copy of the Q-network, parameterized by Q^-. Doing so entails optimizing the DQN Loss function:

$$L_t(\theta_t) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta_t)\right)^2\right]$$

Here:

- $\mathcal{D}$ is a replay memory storing past experiences (s,a,r,s'), sampled uniformly.
- θ^-, the lagging network, is updated periodically from θ to stabilize training.

The weights θ are updated using stochastic gradient descent:

$$\theta_t \leftarrow \theta_t + \alpha \cdot \nabla_\theta L_t(\theta_t)$$

Furthermore, to balance exploration and exploitation, actions a_t are selected using an epsilon-greedy policy:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon_t \\ \arg\max_a Q(s_t,a;\theta) & \text{with probability } 1-\epsilon_t \end{cases}$$

The value of ϵ_t is annealed over time to encourage exploration initially and exploitation later.

3.2 Distributed Architecture

Distributed deep RL approaches address the limitations of traditional RL by parallelizing the learning process. Instead of a single agent interacting with one environment instance, multiple agents (or workers) operate on multiple environment replicas simultaneously. Experience from all these workers is gathered and used to update a central model. This architecture can lead to faster convergence, more efficient exploration, and improved generalization.

Dean et al. [**distbelief**] demonstrated the power of model and data parallelism by introducing the DistBelief framework, which enables the training of deep neural networks with billions of parameters across thousands of machines. Their work demonstrated that both model and data parallelism improve the performance of deep reinforcement learning methods; model parallelism being the distribution of computation tasks across many compute nodes, and data parallelism being the replication of model instances across multiple nodes, allowing distinct subsets of training data to be processed concurrently. By leveraging these strategies, Dean et al. significantly reduced training times while scaling to massive models, enabling state-of-the-art results on tasks such as ImageNet classification and speech recognition. These techniques are directly applicable to deep reinforcement learning, where scaling up training through parallelism is critical for handling the high computational demands of large-scale environments and complex policies.

3.3 GORILA - General Reinforcement Learning Architecture

Following DistBelief came the General Reinforcement Learning Architecture (GORILA) [**gorila**] representing one of the earliest efforts to scale Deep Q-Learning (DQL) beyond the single-agent, single-machine setup. By distributing the learning process across multiple nodes, GORILA leverages parallelism to accelerate training, improve exploration, and ultimately achieve better performance in complex reinforcement learning tasks.

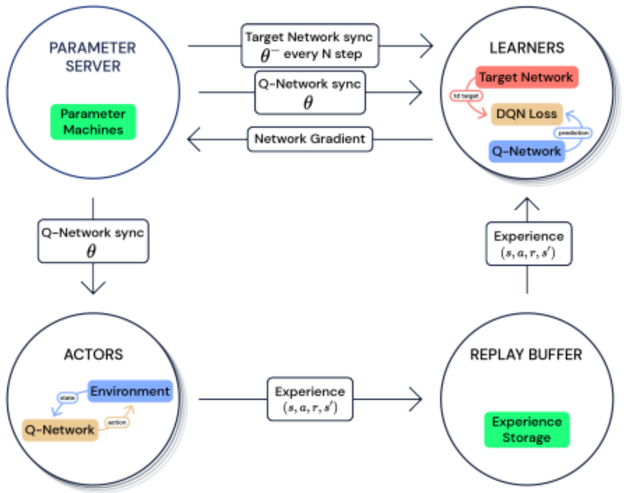

Figure 1: *GORILA Architecture Overview. Multiple actors and learners operate in parallel to generate experiences and update parameters asynchronously, as described by [***DistributedDeepRLOverview***].*

Core Components of GORILA:

- **Actors:** Traditional DQL setups rely on a single agent interacting with the environment, but GORILA deploys N actors $(a_1, a_2, \ldots, a_N)$ in parallel. Each actor is an independent process, maintaining its own local copy of the Q-network Q_i^-. These actors follow a behavior policy (often ϵ-greedy) to select actions and gather experiences (s, a, r, s'). By running multiple actors simultaneously, GORILA ensures a high throughput of experience and broader coverage of the state-action space, thus enhancing exploration and mitigating the risk of converging to suboptimal policies due to limited experience.
- **Experience Replay:** Similar to single-agent DQN, GORILA relies on experience replay buffers for training stability and sample efficiency. However, with multiple actors generating data in parallel, GORILA can maintain either:
  1. A global, centralized replay memory $\mathcal{D}$, which aggregates experiences from all actors into a single, large repository.
  2. Individual, per-actor replay memories that feed into learners separately.

  The global replay memory scales independently of the number of actors, enabling vast amounts of data to be stored and sampled from. This flexibility in memory configuration helps GORILA handle significantly more experiences than single-threaded DQL setups.
- **Learners:** GORILA introduces multiple learners $(L_1, L_2, \ldots, L_M)$, each responsible for performing gradient updates on the Q-network parameters. Each learner pulls mini-batches of experiences from the chosen replay memory (either global or local) and computes gradients g_i with respect to the Q-network parameters. These learners operate asynchronously and in parallel, collectively accelerating the training process. By distributing the workload of gradient computation across multiple learners, GORILA alleviates the computational bottleneck encountered when a single learner attempts to process all experiences sequentially.
- **Parameter Server:** Central to GORILA's architecture is the parameter server, which maintains a global set of Q-network parameters $Q(s, a; \theta^+)$. Unlike a single machine updating parameters in isolation, GORILA shards the parameter vector θ^+ across multiple parameter server nodes 2. Each shard handles a subset of parameters and applies asynchronous gradient updates received from the learners via asynchronous stochastic gradient descent (SGD). This design allows scalable model parameter management and mitigates contention in parameter updates.
- **Asynchronous Updates and Bundled Mode:** One of GORILA's key contributions is its asynchronous communication pattern. Learners compute gradients independently and send them to the parameter server without waiting for synchronous "locks" or synchronization barriers. Actors periodically refresh their local Q_i^- networks by pulling updated parameters from the server. This asynchronous loop—actors pushing experiences, learners pulling data and pushing gradients, and the parameter server updating parameters—significantly boosts throughput and can lead to faster convergence in practice.

  While GORILA supports various deployment configurations, a common approach is the *bundled mode*, where each machine hosts an actor, a replay buffer, and a learner. In this setup, each machine operates as a self-contained unit that can scale horizontally. As more compute resources become available, simply adding more such bundles increases the system's capacity.

Performance and Legacy: GORILA proved instrumental in extending DQL methods to large-scale distributed settings. By applying its architecture to the Atari 2600 benchmark, the authors demonstrated substantial speedups over single-processor implementations, reduced training times, and improved performance. Notably, GORILA achieved human-level or better scores on many Atari games that had previously been challenging for single-agent DQN.

This breakthrough in distributed RL not only showcased the scalability and potential of DQL-based approaches but also laid the groundwork for more advanced distributed reinforcement learning algorithms. Subsequent systems, such as IMPALA, built upon GORILA's insights into parallel exploration, asynchronous updates, and flexible parameter management, further pushing the boundaries of what could be achieved with large-scale reinforcement learning.

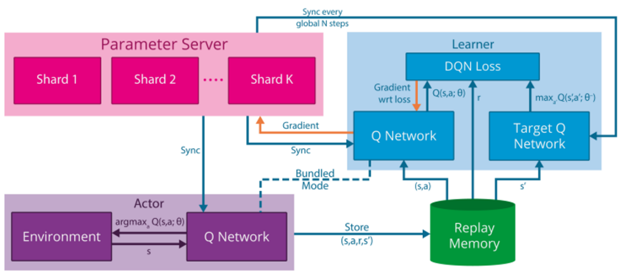

Figure 2: *Distributed Setup of the GORILA Agent [***gorila***]. Multiple learner processes send gradients to the parameter server and receive updated parameters, while independent actors gather experiences in parallel.*

3.4 A3C - Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) [**a3c**] is another prominent distributed reinforcement learning framework that leverages parallelization to accelerate learning. Unlike DQL-based methods that primarily rely on Q-learning and replay memories, A3C adopts a policy gradient approach combined with a value function baseline—hence the term *actor-critic*. The central idea is to run multiple worker agents in parallel, each interacting with its own instance of the environment, and asynchronously updating a shared global model.

Core Concepts of A3C:

- **Actor-Critic Paradigm:** A3C maintains two neural networks that often share initial layers: an *actor* network that outputs a stochastic policy $\pi(a|s;\theta)$ for selecting actions, and a *critic* network that estimates the value function $V(s;\theta_v)$, providing a baseline to reduce the high variance common in policy gradient methods. By doing so, A3C directly optimizes the policy parameters without having to derive them from estimated Q-values.
- **Parallel Actor-Learners:** Instead of a single agent sequentially interacting with the environment, A3C employs multiple *actor-learners* that each run in a distinct thread or on separate CPU cores. Each actor-learner operates independently, selecting actions according to its local copy of the policy, and collecting trajectories (s_t, a_t, r_t, s_{t+1}).
- **Asynchronous Updates:** At periodic intervals, each actor-learner computes gradients for both the policy and the value function using the data it collected, and then applies these gradients to a globally shared set of model parameters. Unlike DQL methods that rely on a replay buffer and synchronized target networks, A3C's global parameters are updated *asynchronously*, removing the need for expensive locking mechanisms or single-threaded bottlenecks. The asynchronous updates help to de-correlate the training samples without explicitly using a replay memory.
- **Advantage Estimation:** A central component of A3C's stability and performance is the use of the *advantage* function:

$$A(s_t, a_t; \theta_v) = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}; \theta_v) - V(s_t; \theta_v)$$

  This advantage function quantifies how much better (or worse) an action is compared to the baseline expected return. By using advantages rather than raw returns or Q-values, A3C reduces variance in the gradient estimates and encourages more stable and efficient learning.
- **On-Policy Learning:** Unlike GORILA and other distributed Q-learning methods that rely on replay buffers, A3C is inherently an on-policy method. The trajectories used to update the global parameters come directly from the actor-learners' current policy. Although it forgoes the stabilizing effects of replay memory, the parallelization across many different environments and initial conditions provides an implicit form of regularization and decorrelation.

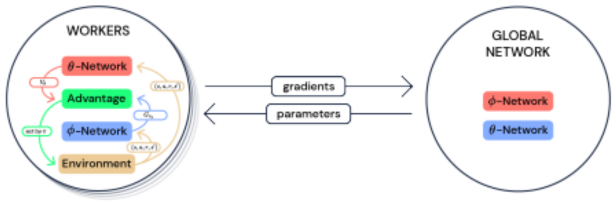

Figure 3: *A3C Architecture Overview. Multiple actor-learners run asynchronously, each updating the global model. Adapted from [***a3c***].*

Training Procedure: At a high level, the A3C training loop proceeds as follows:

1. *Initialize Global Parameters:* Start with a global policy and value network with parameters θ and θ_v.
2. *Spawn Actor-Learners:* Multiple actor-learners run in parallel. Each one:
   (a) Copies the global parameters into local parameters θ', θ'_v.
   (b) Interacts with its environment, taking actions a_t sampled from $\pi(a|s;\theta')$ and collecting rewards r_t until it reaches an update interval or a terminal state.
   (c) Uses the collected experience to compute gradients ∇_θ and ∇_{θ_v} based on the policy gradient and value loss (plus an entropy regularization term to encourage exploration).
   (d) Atomically updates the global parameters:
$$\theta \leftarrow \theta + \alpha \nabla_\theta; \quad \theta_v \leftarrow \theta_v + \alpha_v \nabla_{\theta_v}$$
   (e) Resets its local parameters to the updated global parameters and continues interacting with the environment.

Key Advantages of A3C:

- **Increased Training Speed:** By leveraging multiple CPUs, A3C significantly reduces training time compared to single-threaded methods. Parallel workers ensure a steady flow of fresh data, reducing the waiting time that a single-agent setup would incur.

- **Stability and Robustness:** The combination of asynchronous updates and advantage-based gradients helps stabilize learning. Parallel exploration mitigates issues associated with local optima, mode collapse, or insufficient coverage of the state-action space.
- **Simplicity of Infrastructure:** Unlike architectures that require a parameter server or distributed replay memory, A3C relies on parallel threads and a globally shared model. This reduces the overhead and complexity of system design, making it relatively straightforward to implement and scale on modern multi-core CPUs.

Performance and Impact: A3C's design allowed for near-linear speedups in training time as the number of parallel workers increased. When applied to challenging domains such as Atari games or 3D navigation tasks, A3C achieved strong performance that was competitive with or surpassed contemporaneous methods. The insights from A3C's asynchronous and advantage-based learning informed subsequent distributed RL techniques, many of which combine A3C's simplicity with more advanced architectures, policies, and exploration strategies.

Ultimately, A3C showcased how distributing computation and employing an actor-critic approach could push the boundaries of what was computationally feasible in reinforcement learning, inspiring a host of future algorithms that continue to push RL scalability and efficiency.

3.5 IMPALA - Importance Weighted Actor-Learner Architecture

The Importance Weighted Actor-Learner Architecture (IMPALA) [**impala**] is a distributed reinforcement learning framework that builds on the lessons learned from earlier systems like GORILA and A3C, further refining the balance between scalability, stability, and sample efficiency. IMPALA introduces a novel off-policy correction method, known as V-trace, which allows for efficient training with large numbers of actors while preserving stable convergence properties.

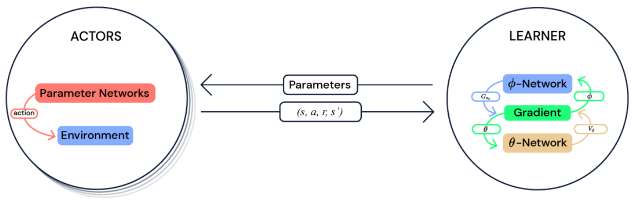

Figure 4: *IMPALA Architecture Overview. Multiple actors generate experience in parallel and send data to a central learner that updates model parameters, inspired by [***DistributedDeepRLOverview***].*

Key Features of IMPALA:

- **Decoupled Actor-Learner Architecture:** IMPALA cleanly separates the processes that generate experience (actors) from the process that computes parameter updates (the learner). The actors run continuously and asynchronously, each interacting with its own environment instance. They execute a parameterized policy $\pi(a|s;\theta_{\text{actor}})$ to collect trajectories of experience, which are then sent to a centralized learner.
  Unlike A3C, where each actor-learner thread updates the global parameters directly, IMPALA's actors do not perform gradient updates. Instead, they offload all learning responsibilities to a single, more powerful learner node (or a small set of such nodes). This architecture enables more efficient use of computational resources, as the learner can leverage hardware accelerators (e.g., GPUs) for batch updates without contending with actor processes for shared resources.
- **Off-Policy Corrections (V-trace):** A significant innovation in IMPALA is the V-trace off-policy correction method. Because actors may lag behind the current learner policy (due to communication delays and asynchronous updates), the trajectories they produce do not exactly match the learner's current policy. Without correction, this discrepancy can bias the gradient updates, destabilizing learning.
  V-trace addresses this issue by reweighting returns to approximate on-policy updates even when behavior and target policies differ. Specifically, V-trace modifies the value targets using truncated importance sampling ratios:
  $$v_s = V(s;\theta_v) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i \right) \delta_t$$
  where δ_t is a TD-error-like term and c_i are truncated importance weights. By carefully adjusting these weights, V-trace ensures that the learner's updates remain stable and efficient, even as the number of actors and the staleness of their parameters grow.
- **Scale and Efficiency:** IMPALA's design is geared toward extremely large-scale scenarios. By removing the need for actors to compute gradients, it reduces overhead and streamlines the data pipeline. Thousands of actors can be run in parallel, each generating experience and sending it asynchronously to the learner. This allows IMPALA to leverage massive compute clusters and dramatically speed up training, while V-trace maintains performance quality.
  On a single machine, using multiple CPU cores or threads, IMPALA still brings considerable benefits by parallelizing experience collection and training steps. This dual capability allows it to serve both as a high-throughput, industrial-strength RL solution and as a convenient prototyping platform for researchers and practitioners without access to large-scale computing infrastructure.
- **Flexible Policy Class:** IMPALA is not tied to a specific RL algorithm, such as Q-learning or policy gradients. Instead, it supports a broad class of policy optimization methods (often actor-critic styles). This flexibility makes it possible to integrate more advanced architectures, auxiliary tasks, or hierarchical policies. By doing so, researchers and practitioners can tackle a wide range of environments and domains.

Training Procedure: The IMPALA training loop involves the following steps:

1. *Actor Processes:* A large number of actor processes run

in parallel. Each actor maintains a local copy of the policy parameters, selects actions, and collects experience sequences (trajectories). These trajectories include states, actions, rewards, and policy logits or probabilities.

2. *Learner Node:* The learner receives trajectories from all actors. It uses V-trace to compute off-policy corrected targets and applies updates to the model parameters via stochastic gradient descent. The learner is typically equipped with GPUs or TPUs for fast, large-batch learning.

3. *Parameter Updates:* After the learner updates the shared model parameters, the updated parameters are asynchronously broadcast back to the actors. Actors periodically refresh their local parameters, ensuring that they never deviate too far from the learner's current policy.

Performance and Impact: IMPALA demonstrated strong empirical results on a variety of challenging RL benchmarks, including Atari and multi-agent environments. It achieved near-linear scaling in performance as more actors were added and often reached comparable or superior results faster than previous distributed methods.
Its introduction of the V-trace algorithm provided a general and elegant solution for handling off-policy corrections in large-scale distributed settings. IMPALA's architectural decisions—centralizing learning, employing stable off-policy corrections, and supporting a wide range of policies—significantly influenced subsequent distributed RL frameworks and remains a leading choice for tackling computationally intensive RL problems.
In essence, IMPALA's approach to scaling reinforcement learning combined the best ideas from predecessors like A3C and GORILA, while introducing innovative solutions like V-trace. This resulted in a robust, flexible, and highly scalable system that continues to guide the development of next-generation distributed RL algorithms.

4 Implementation Details

4.1 Tools and Rationale

For our distributed deep reinforcement learning approach to path finding, we chose to utilize the IMPALA algorithm [**impala**]. IMPALA strikes an ideal balance between scalability, flexibility, and stable learning dynamics. It can run efficiently on a single machine with multiple CPUs or GPUs, which makes it convenient for our resource constraints—essentially a "distributed" setup in miniature. Simultaneously, if more computational resources become available, IMPALA can seamlessly scale up to operate across thousands of machines, handling massive parallelization without extensive re-engineering.
Another key advantage of IMPALA is its off-policy correction mechanism, V-trace, which ensures stable updates even when there is a lag between actor policies and the central learner's parameters. This capability is particularly beneficial in distributed settings, where asynchronous updates are the norm. Furthermore, IMPALA is considered a conceptual successor to both A3C and GORILA, inheriting the simplicity of A3C's architecture and the scalability of GORILA's multi-machine approach. By leveraging IMPALA, we anticipate more efficient training and potentially improved performance compared to either predecessor—making it an ideal choice for our proof of concept.
To implement IMPALA, we rely on RLlib [**rllib**], a scalable reinforcement learning library built on top of Ray [**ray**], which simplifies the engineering overhead of parallelization. RLlib's built-in support for IMPALA provides ready-to-use training scripts, hyperparameter configurations, and monitoring tools, enabling rapid experimentation and iteration.

4.2 Custom Environments with Gymnasium

Our experiments were conducted using custom environments built with *Gymnasium*, a community-driven successor to OpenAI Gym. Each environment represents a path planning scenario on a small 2D grid (e.g., 2×2 or larger variations), where the agent starts at a designated initial location and must navigate towards a goal state while avoiding obstacles. The state space typically includes the agent's current position and any partial observations of the surrounding grid. Rewards are assigned to encourage efficient navigation:

- **Positive Reward:** Received when the agent successfully reaches the goal.
- **Negative Reward:** Imposed for collisions with obstacles or invalid actions (e.g., moving into a wall).
- **Step Penalties:** Small negative rewards per step to discourage aimless wandering and encourage efficient paths.

By using Gymnasium, we ensure that our environment abides by a standard RL interface, supporting extensibility and reproducibility. This modularity allows us to easily substitute different environment configurations or even more complex path planning scenarios without altering the training pipeline.

4.3 Using RLlib for Distributed Training

We leverage RLlib's native IMPALA implementation to manage the training lifecycle. RLlib seamlessly handles:

- **Actor Management:** Spawning multiple parallel actor processes (or threads) that collect experience from independent environment instances.
- **Learner Updates:** Aggregating trajectories from actors, applying V-trace off-policy corrections, and performing asynchronous parameter updates on a centralized model.
- **Hyperparameter Tuning and Logging:** Facilitating experiments with different learning rates, exploration parameters, or neural network architectures. RLlib integrates with visualization tools (e.g., TensorBoard) to monitor metrics like episode returns, policy entropy, and value function loss over time.
- **Resource Allocation:** Running on a single machine or distributing computation across multiple nodes if necessary. RLlib, built on Ray, can scale from a laptop to a cluster without major code changes.

By closely observing these metrics, we gain insights into the policy's quality, the training framework's scalability, and the effectiveness of IMPALA's distributed learning process. This holistic evaluation allows us to not only benchmark our approach against single-agent and non-distributed baselines but also highlights the potential of deploying IMPALA in resource-constrained settings.
All experiments use RLlib's default IMPALA hyperparameters. Actors and the central learner run on a single machine with multiple CPU cores, simulating a miniature distributed setup. The agent receives positive rewards upon successfully reaching the goal and incurs small penalties for collisions and unnecessary steps, encouraging efficient navigation.
In sum, by combining IMPALA's scalable distributed learning algorithm with Gymnasium's flexible environment interface and RLlib's robust tooling, we have constructed a practical, end-to-end system for investigating distributed deep reinforcement learning applied to the path planning problem.

5 Experiments

5.1 Single Grids with No Moving Obstacles

In our first experiment, we begin by training IMPALA to solve a static 10×10 2D grid environment with no moving obstacles. This initial scenario serves as a baseline to understand IMPALA's learning dynamics under controlled conditions.
We consider two configurations of the 10×10 grid: an *Easy* configuration, where the goal is relatively straightforward to reach, and a *Hard* configuration, where the agent must navigate a more winding path to reach the target. Figures 5 and 6 visualize the two environments, each annotated with one of the known optimal paths (in blue) for illustrative purposes.
Figure 7 shows the mean episode reward over training iterations for the Easy configuration. The agent converges to a near-optimal policy in roughly 15 training iterations. Similarly, the Hard configuration (Figure 8) also converges in about the same number of iterations, demonstrating that path complexity—defined by the minimum path length or number of required turns—does not significantly affect IMPALA's convergence speed in these small-scale settings.
To further probe the impact of environment size on learning speed, we introduce a 5×5 Hard configuration (Figures 9 and 10). Here, the agent converges even more quickly, often stabilizing within 10 iterations (33% increase). This suggests that environment size plays a more substantial role in learning speed than path complexity alone. Smaller grids contain fewer states and possible trajectories, allowing the agent to more quickly and thoroughly explore the state-action space.
From this initial set of experiments, we conclude:

- IMPALA can efficiently learn to navigate and solve basic path planning tasks on small, static grids, converging within a handful of training iterations.
- Increasing path complexity does not drastically slow learning, suggesting that IMPALA is capable of handling a variety of route-finding challenges.
- Environment size has a more pronounced effect on convergence speed, with smaller environments yielding faster training times.

Next, we experimented training a single model on multiple environments, each with a different configuration of obstacles. There are 2 general cases to examine: using a model on environments **with a common optimal path** and using it on environments **without a common optimal path**. This distinction is critical for understanding the model's ability to generalize across diverse setups.

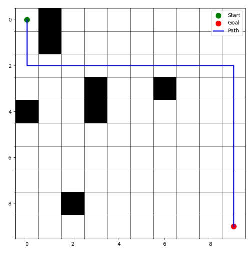

Figure 5: 10×10 *Grid (Easy). The blue line indicates a known optimal path.*

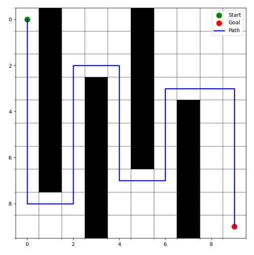

Figure 6: 10×10 *Grid (Hard). The blue line indicates a known optimal path.*

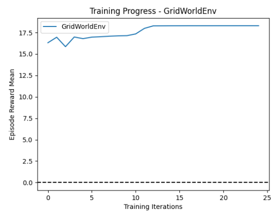

Figure 7: *Mean Episode Reward for the* 10×10 *Easy Environment, showing rapid convergence.*

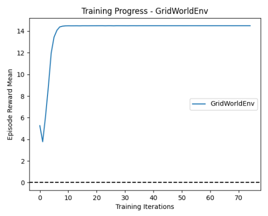

Figure 8: *Mean Episode Reward for the* 10×10 *Hard Environment, also converging quickly.*

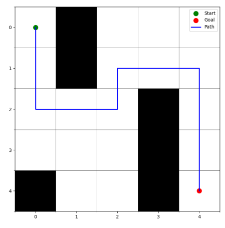

Figure 9: 5×5 *Grid (Hard). The blue line represents a known optimal path.*

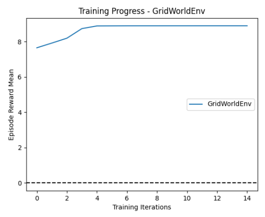

Figure 10: *Mean Episode Reward for the* 5×5 *Hard Environment, converging even faster.*

5.2 Multiple Different Grids With a Common Optimal Path - Trivial Approach

To establish a baseline for our multi-environment training experiments, we first employed a trivial approach: training the model from scratch until convergence on each environment in sequence. In this method, the model is fully trained on one environment before being moved to the next, without any attempts to simultaneously leverage knowledge gained from previously trained environments.
In the presence of a common optimal path between multiple environments, the trivial sequential training approach converged remarkably quickly and retained solutions that worked

for both tasks. Because the solution strategies overlapped, the model did not have to unlearn previously optimal behaviors when transitioning from one environment to the next. As a result, both environments were solved efficiently, and the model's final policy generalized to both tasks without significant performance degradation.

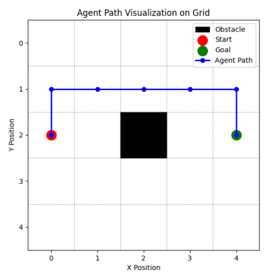

Figure 11: *Policy learned on Environment 1 after sequential training on two environments sharing a common path.*

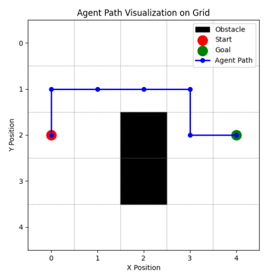

Figure 12: *Policy learned on Environment 2, showing successful transfer from Environment 1 due to shared optimal paths.*

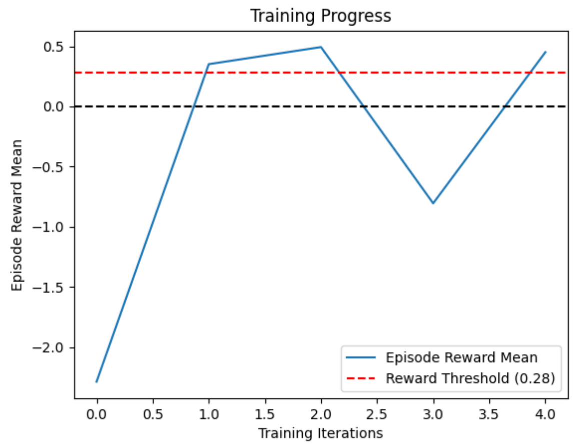

Figure 13: *Convergence Graph for Sequential Training on Two Environments with a Common Optimal Path. Minimal performance drop after introducing the second environment.*

Moreover, the minimal drop in mean reward observed after introducing a second environment suggests that the model effectively leveraged knowledge acquired from the first task. This form of transfer learning allowed it to quickly adapt and exploit the similarities between the two environments, accelerating convergence and maintaining high overall performance. This initial experiment demonstrates that when environments share a very similar underlying solution or route structure, simply retraining the model sequentially on each environment is sufficient to achieve stable and rapid convergence. However, this trivial approach relies heavily on the existence of a shared optimal path. Next, we examine the scenario where no such common solution exists.

5.3 Multiple Different Grids Without a Common Optimal Path - Trivial Approach

To assess the limitations of sequential training, we repeated the trivial approach on a set of environments that do not share a common optimal path. In this case, the solution that works well for one environment may be suboptimal or even invalid for another. Our goal was to understand how the model adapts when forced to discard previously learned navigation strategies and adopt new ones.

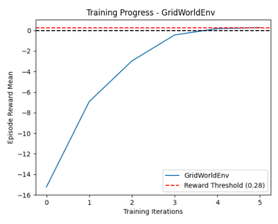

Figure 14: *Mean Reward for Environment 1 trained alone: rapid convergence.*

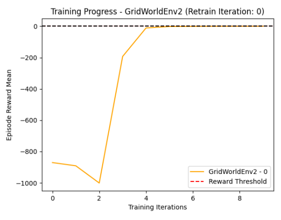

Figure 15: *Mean Reward when retraining on Environment 2 after Environment 1. Initial drop due to conflicting strategies.*

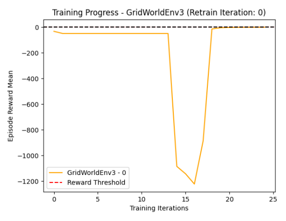

Figure 16: *Mean Reward for Environment 3 after exposure to Environments 1 and 2. Convergence is much slower.*

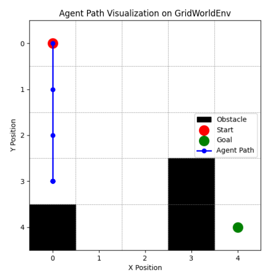

Figure 17: *Trajectory in Environment 1 after final training on Environment 3, showing catastrophic forgetting of previously learned solutions.*

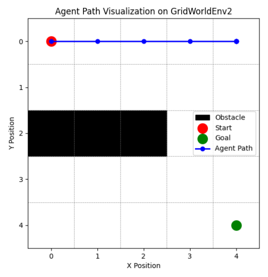

Figure 18: *Trajectory in Environment 2 after training on Environments 1 and 3. Previously optimal solutions are not retained.*

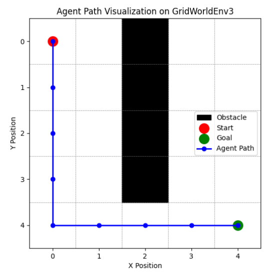

Figure 19: *Trajectory in Environment 3 after training on Environments 1 and 2. The agent converges here, but older tasks are forgotten.*

In this scenario, the model only converges on the final environment seen. Upon introducing each new environment, the agent attempts to reuse previously learned solutions, resulting in initially high losses and a drop in mean episode reward. Over time, it unlearns the old solution and adopts a new one

specific to the latest environment. The cost of this adaptation grows with each additional environment. Notably, by the time the agent is exposed to the third environment, convergence takes substantially longer. Furthermore, the agent forgets how to solve previously mastered tasks, indicating a form of catastrophic forgetting. This phenomenon is not unexpected: without a shared policy that benefits all environments or a mechanism to retain old knowledge, the agent's final policy collapses to a solution tailored only to the last environment trained.
These findings highlight the fundamental limitation of the trivial sequential training approach when environments are distinct and lack a common optimal strategy. To address this challenge, more sophisticated methods—such as multi-environment training or continual learning techniques—are necessary.

5.4 Multiple Different Grids with a Common Optimal Path - Exhaustive Iterative Multi-Environment Training

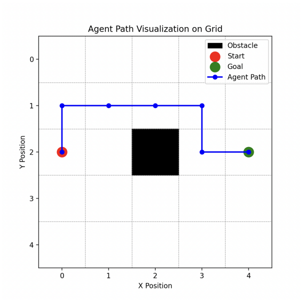

Figure 20: *Environment 1 Trajectory after iterative training on Environments 1 and 2. The model finds a robust common path.*

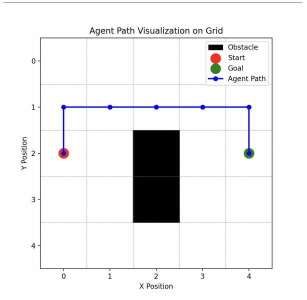

Figure 21: *Environment 2 Trajectory after iterative training with Environment 1. Policy generalizes to both tasks.*

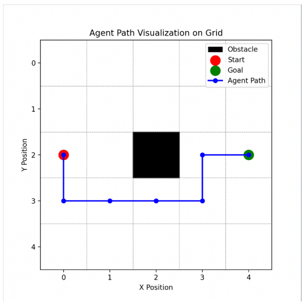

Figure 22: *Environment 1 Trajectory after training on Environments 1 and 3, also converging on a common solution.*

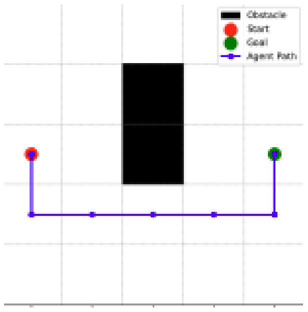

Figure 23: *Environment 3 Trajectory after iterative training with Environment 1. The shared path structure aids convergence.*

Motivated by the shortcomings of the trivial approach, we turn to a more iterative training methodology. Here, the goal is to exploit the presence of a shared optimal path across multiple environments by training until conversion each one in an rotating fashion multiple times rather than simply converging on each one once.

To test our new hypothesis, we alternately trained a single model on a pair of environments, each for 200 iterations, cycling between the two environments multiple times. The idea is that the model has the opportunity to converge on a stable solution in one environment, then re-validate and adjust its policy on the other environment without losing previously learned solutions. We did so twice: once with environment 1 & 2 and another time on environment 1& 3.
Figures 20 and 22 show two viable optimal paths for environment 1. The agent can navigate around obstacles by moving either upward or downward, offering multiple equally good paths. Figures 21 and 23 show the optimal paths of envs 2 and 3 respectively.
This iterative, alternating training strategy proved successful in the presence of a shared optimal path. Both pairs of environments (1 & 2, and 1 & 3) converged on solutions that worked for all tasks involved. By frequently revisiting each environment, the model reinforced a stable policy that did not degrade when exposed to a different, yet compatible, environment. This suggests that when a common solution exists, regular alternating updates between tasks can yield robust policies that generalize well.

5.5 Multiple Different Grids Without a Common Optimal Path - Exhaustive Iteration Multi-Environment Training

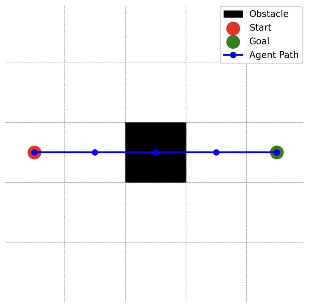

Figure 24: *Environment 1 demands a distinct route that does not generalize to others.*

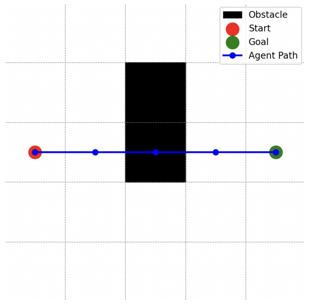

Figure 25: *Environment 2 requires a different solution, conflicting with Environment 1's.*

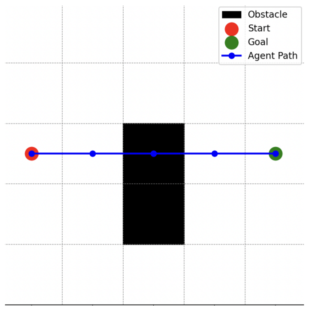

Figure 26: *Environment 3's optimal path is also incompatible, preventing a universal policy.*

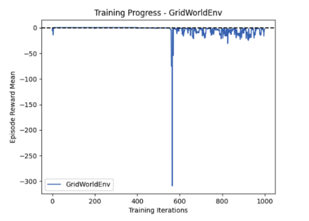

Figure 27: *Convergence graph for three distinct environments. Persistent oscillations indicate failure to find a common solution.*

Having established that iterative multi-environment training can work well when environments share a common optimal path, we now turn to a scenario in which no such overlap exists. In this case, we applied the same alternating training

strategy to three distinct environments, none of which share a unifying route structure.
Figures 24, 25, and 26 illustrate that each environment demands a distinct optimal path. With 200 training iterations allocated to each environment before switching, the model would repeatedly converge on environment-specific solutions, only to forget them when exposed to a new, structurally incompatible environment. As shown in Figure 27, this leads to an initial reward drop followed by persistent oscillations. The model continually re-adapts to each environment but never settles on a universal policy, highlighting the difficulties of generalization when no common solution exists.
This outcome emphasizes that while iterative multi-environment methods can leverage overlapping structures to produce stable, generalizable policies, they struggle when each task demands a fundamentally different strategy. The agent becomes trapped in a cycle of catastrophic forgetting, overfitting to the currently active environment and then failing to transfer that knowledge effectively to the others.
To address this challenge and mitigate overfitting, we next explore how altering the duration of training on each environment affects the model's ability to generalize. By reducing the number of iterations per environment, we aim to prevent the model from becoming overly specialized before encountering a new task, potentially improving its overall stability and performance across all environments.

5.6 Multiple Different Grids Without a Common Optimal Path — Revolving Mini-Batch Training

In light of the challenges encountered when using large, fixed training intervals on each environment, we introduce a *Revolving Mini-Batch* approach. Instead of committing to hundreds of iterations per environment before switching, we greatly reduce the number of training steps each environment receives at a time and cycle through them more frequently. By rapidly rotating between environments with minimal per-environment training, we aim to prevent the model from overfitting to any one task and encourage more balanced, generalizable learning across all environments.

5.6.1 30-Iteration Revolving Mini-Batch

We first reduced the per-environment training interval to 30 iterations before rotating to the next task. This change was motivated by the hypothesis that shorter training periods would limit overfitting and reduce the formation of suboptimal, environment-specific policies.
This adjustment yielded promising results. Compared to the previous 200-iteration scenario, the model displayed far fewer oscillations and demonstrated greater stability in the convergence graph (Figure 31). The agent trajectories in Figures 28, 29, and 30 also show that the agent is less likely to commit to a single, suboptimal route across all environments. Instead, it learns more nuanced, environment-specific strategies, better reflecting each environment's unique constraints.
However, while this reduced training interval diminished overfitting and improved performance in the first two environments, the model still struggled to find the optimal policy in the third environment. The rewards associated with that environment remained consistently lower. Thus, although shorter training cycles led to more balanced learning, achieving full generalization to all tasks remained elusive under these conditions.

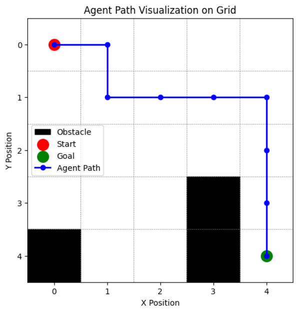

Figure 28: *Environment 1 Trajectory with 30-iteration revolving training. Less overfitting observed.*

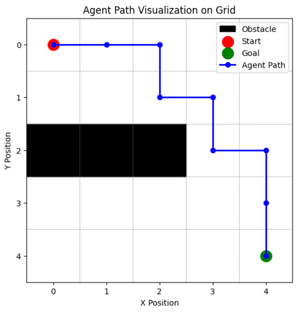

Figure 29: *Environment 2 Trajectory with 30-iteration revolving training. Stability improves.*

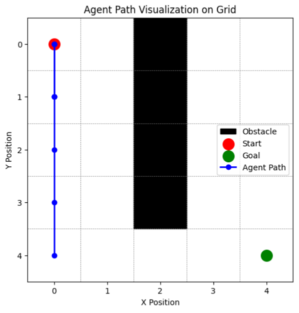

Figure 30: *Environment 3 Trajectory with 30-iteration revolving training. Improvements, but not perfect.*

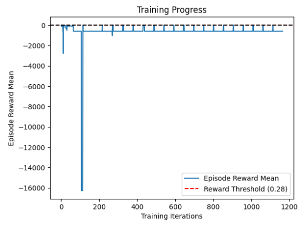

Figure 31: *Convergence graph for 30-iteration revolving training, showing fewer oscillations but still struggles with Environment 3.*

5.6.2 2-Iteration Revolving Mini-Batch

Encouraged by the results of the 30-iteration approach, we pushed the idea further by reducing the training interval to just 2 iterations per environment before rotating.
Remarkably, this drastic reduction further mitigated overfitting. As shown in Figures 32, 33, and 34, the model successfully adapted to all three environments without collapsing into suboptimal, all-purpose strategies. The convergence graph in Figure 35 reflects this improved balance—there are no severe drops, and oscillations are minimal.
In addition to enhancing stability, the 2-iteration approach significantly accelerated convergence. By never allowing the model to become too comfortable in a single environment, we effectively promoted a continual "resetting" of its short-term memory and prevented entrenched, environment-specific habits from taking hold. This method proved highly effective in enabling the model to find a workable compromise policy across a diverse set of tasks in a relatively short span of training time.
In summary, the Revolving Mini-Batch strategy demonstrates that carefully tuning the per-environment training intervals can substantially improve model stability, mitigate overfitting, and foster better generalization in complex multi-

environment setups—even when no common optimal path exists. Although achieving perfect generalization in all cases remains challenging, these experiments underscore the potential of rapid environment switching as a tool for balancing competing objectives in distributed reinforcement learning.

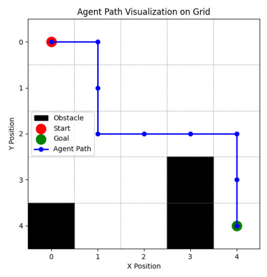

Figure 32: *Environment 1 Trajectory with 2-iteration revolving training. Rapid adaptation observed.*

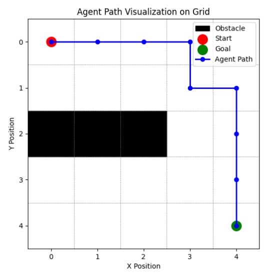

Figure 33: *Environment 2 Trajectory with 2-iteration revolving training. Policy remains stable.*

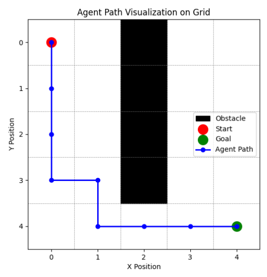

Figure 34: *Environment 3 Trajectory with 2-iteration revolving training. Good balance maintained.*

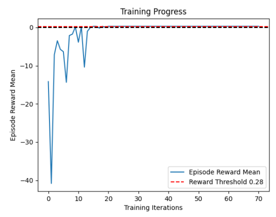

Figure 35: *Convergence graph for 2-iteration revolving training, showing stable performance across all tasks.*

5.7 Multiple Grids with Varying Start and Destination Points

In the previous experiments, the locations of the start and goal states remained fixed within each environment. Next, we explore a more challenging setting: multiple environments where not only the grid layouts differ, but the starting and ending points of each navigation task also vary. This scenario pushes the model to generalize beyond a static route and adapt to new initial and terminal states on the fly.

When we introduced these more dynamic conditions, the training process became noticeably more volatile. As seen in Figure 38, the mean episode reward experiences large early drops—reaching as low as -1400 at iteration 5 and around -700 near iteration 8. These sudden declines suggest that when the environment switches, the model initially struggles to adapt to new start-goal configurations and may temporarily rely on previously learned, now-inappropriate strategies. This difficulty is analogous to the overfitting and catastrophic forgetting behaviors observed earlier. Despite the challenging nature of this setup, the model eventually stabilizes and converges. By around 25 or more total training iterations, the mean reward settles at a level indicating that the agent has successfully internalized policies that can handle the diversity of layouts and start-end point placements. Figure 36 and Figure 37 illustrate the different grid configurations, emphasizing that neither the obstacles nor the start and end points remain constant between environments.

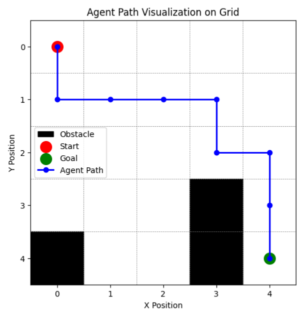

Figure 36: *Environment 1 with varied start/end points, increasing complexity.*

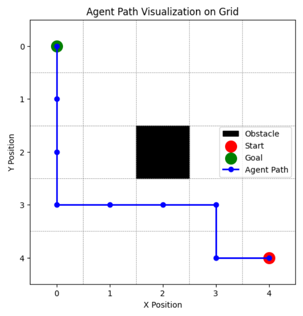

Figure 37: *Environment 2 features distinct obstacles and flipped start/goal states relative to Environment 1.*

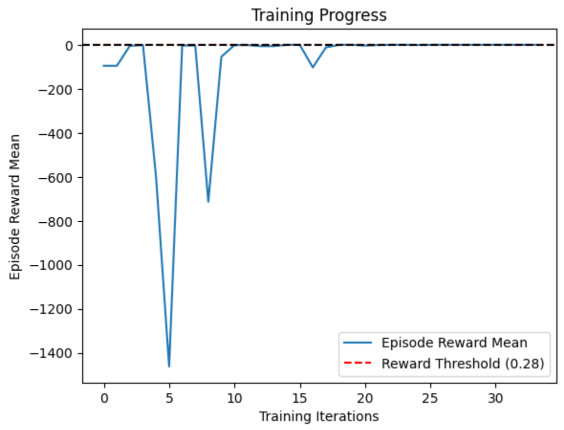

Figure 38: *Convergence graph for varying start/end points. Early instability is followed by eventual convergence after 25 iterations.*

This outcome demonstrates that our distributed, iterative, and flexible training approach can extend beyond fixed start-goal pairs. Even with the increased complexity of dynamically changing initial and terminal states, the model's learning process ultimately adapts. While the early stages of training are marked by sharp drops in performance, consistent cycling and reduced iteration phases (as previously explored) enable the system to recover, mitigate overfitting, and find stable navigation policies that operate effectively across multiple, distinct starting and ending conditions.

6 Conclusion

In this paper, we presented a scalable approach to path planning through distributed deep reinforcement learning, specifically using the IMPALA algorithm. Our experiments validated that IMPALA can efficiently learn navigation policies across multiple environments, even under challenging conditions such as differing obstacle layouts, non-overlapping optimal paths, and dynamically changing start and end points.

A key insight from our work is the effectiveness of shortening per-environment training intervals—an approach we termed the revolving mini-batch strategy. By reducing the number of training iterations before switching to another environment, we significantly mitigated overfitting and catastrophic forgetting. This adjustment allowed a single policy to balance the constraints of multiple environments, maintaining stability and improving overall performance. While some scenarios remained challenging—especially when no common solution existed—rapid cycling still yielded more robust and generalizable policies than traditional, long-interval training.

These findings suggest that distributed RL frameworks can fully exploit the parallel computational power inherent in modern hardware and the expanding IoT ecosystem. Beyond the current scope, our methods can be extended by integrating classical path planning solutions like A* to establish baseline policies or guide exploration. For instance, one could dynamically decompose large, complex environments into manageable subproblems (e.g., 5×5 grids), solve them iteratively, and recombine solutions to handle large-scale and dynamic tasks in real-time.

In conclusion, by leveraging distributed RL, careful training interval tuning, and off-policy corrections, we have taken a step toward more adaptive, scalable, and efficient path planning solutions. Future research can build on these concepts, combining them with hierarchical RL or curriculum learning, and exploring their application in increasingly complex, real-world navigation scenarios.

7 Contributions

We've both worked equally on every section.

8 Appendix

8.1 Reward System

Throughout the experiments, the way we determined the reward threshold was via a simple formula which empirically worked well and was fine tuned as we made different experiments:

$$\text{threshold} = \text{goal reward} + \text{min path length} \times \text{step reward penalty} - \epsilon$$

where ϵ was a hyperparameter.

8.2 Technical Difficulties

- Ray RLlib 2.39.0 is significantly different from 2.4.0 and many examples referenced in the documentation is out of date. It is best to look through the source code, but that is only for 2.4.0.

8.3 A* path finding and deep reinforcement learning

Since generalization over an environment that an RL model was not trained on is out of reach, we have decided to come up with a new algorithm utilizing classic path-finding algorithms such as A* and combine it with our reinforcement learning model. Given the fact that we can exhaustively train the model on a smaller-size grid, we can utilize it to solve optimal collisions and drive a sub-optimal path toward the goal.

1. Run the A* algorithm and find the optimal path at that specific snapshot of the environment.
2. Get the sub-grid of size N around the agent which is supposed to move on the path.
3. Make the agent traverse the path and at each step check if there is a collision $N/2$ step ahead of the agent.
4. Upon finding the collisions $N/2$ steps ahead of the moving agent. set up the $N \times N$ sub grid.
   (a) The start point is the current location of the agent.
   (b) The end point is the coordinate at the end of the grid set by A*.
   (c) Convert the $N \times N$ grid we have to the OpenAI Gym environment and run the pre-train model to find the shortest path on that sub-grid.
   (d) Make the agent follow the path set by the RL model until it has existed the N $\times$ N matrix.
5. set a threshold between the amount of acceptable changes on an environment. describe this threshold as a similarity matrix. if the similarity matrix is larger than the set threshold give up on fixing the path on the spot and run the A* algorithm one more time.

At first we wanted to incorporate this idea of using A* with 3×3 grids. This was due to the fact that there are relatively fewer possible configurations of 3×3 grids to train a model on and we initially believed that generalization in reinforcement learning was a very difficult task. However, when we attempted to train a model for 3×3 environments, we found that it would often overfit very quickly, thus making it unable to converge on multiple environments and even less so solve new environments. Ultimately, this let us try with 3×3 grids which is what we did in the report.

8.4 Path modification

Another possible solution that we explored but ultimately discarded is an RL model that modifies a path. In this solution, given an environment and an initial path proposed, the RL model changes the path minimally to avoid obstacles.

This is the way we approached implementing it:

1. The agent does not traverse the grid anymore; instead, the agent modifies the given path.
2. Step one: Find an optimal path assuming there are no obstacles on the grid.
3. Step two: Identify the collision points of the path with the obstacles.
4. Step three: Add actions `(up, down, left, right)` to the action space for each specific collision point. Train a model to:
    - Interact with the path.
    - Move each segment of the path `up`, `down`, `left`, or `right` while maintaining the continuity of the path.

Action Space: *[(collision 1, up), (collision 1, left), (collision 1, right), (collision 1, down), . . . , (collision n, up), (collision n, left), (collision n, right), (collision n, down)]*

State Space: $grid, path, obstacles$

Reward System

1. Small reward for reducing the number of collisions.
2. Significant reward for reducing the number of collisions to zero.
3. Small penalty if the number of collisions remains the same.
4. Penalty proportional to the increase in path length:

$$\text{Penalty} = (\text{small penalty}) \cdot (\text{new length} - \text{old length})$$

5. Significant penalty for invalid moves or increasing the number of collisions.