HEPPO: HARDWARE-EFFICIENT PROXIMAL POLICY OPTIMIZATION

HEPPO: HARDWARE-EFFICIENT PROXIMAL POLICY OPTIMIZATION
A UNIVERSAL PIPELINED ARCHITECTURE FOR GENERALIZED ADVANTAGE ESTIMATION

By HAZEM TAHA,

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2024)

Hamilton, Ontario, Canada (Department of Electrical and Computer Engineering)

TITLE:	HEPPO: Hardware-Efficient Proximal Policy Optimization A Universal Pipelined Architecture for Generalized Advantage Estimation
AUTHOR:	Hazem Taha B.Eng. (Computer Engineering), Nile University, Egypt
SUPERVISOR:	Dr. Ameer Abdelhadi
NUMBER OF PAGES:	xvii, 128

Lay Abstract

Reinforcement Learning (RL) enables agents to acquire knowledge and make decisions by interacting with their environment, similar to human experiential learning. RL has been widely used in various industrial domains such as health care and finance. However, the implementation of sophisticated RL algorithms on small, resource-limited devices such as embedded systems or edge devices is often difficult due to the fact that they require a significant amount of computational power and memory. This thesis presents HEPPO, a novel framework facilitating the efficient execution of the widely-used reinforcement learning algorithm, Proximal Policy Optimization (PPO), across several hardware platforms. By optimizing critical bottlenecks of the algorithm and developing a customized hardware architecture, HEPPO markedly decreases computational requirements and memory consumption without compromising performance. The proposed framework enables the real-time deployment of intelligent learning algorithms on devices such as drones, robots, and other smart systems, thereby augmenting their capabilities and facilitating new avenues for innovation across diverse industries.

Abstract

This thesis presents HEPPO: Hardware-Efficient Proximal Policy Optimization, a framework designed to address the computational and memory challenges associated with implementing advanced reinforcement learning algorithms on resource-constrained hardware platforms. By introducing dynamic standardization for rewards and an 8-bit quantization strategy, HEPPO reduces memory requirements by up to 75% while improving training stability and performance, achieving up to a 67% increase in cumulative rewards. A novel, highly parallelized architecture for Generalized Advantage Estimation (GAE) computation accelerates this critical phase, processing 19.2 billion elements per second using 64 processing elements, contributing to a 22% to 37% reduction in PPO training time in different environments. Adapting the proposed on-chip memory layout reduces the GAE data transfer latency and increases the reduction percentage up to 48% in certain environments in PPO training time. The integration of the entire PPO pipeline on a single System-on-Chip (SoC) further enhances system performance by reducing communication overhead and leveraging custom hardware acceleration. Experimental evaluations demonstrate that HEPPO effectively bridges the gap between sophisticated reinforcement learning algorithms and practical hardware implementations, enabling efficient deployment in embedded systems and real-time applications.

To my family and friends,

for their unwavering support and encouragement.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Ameer Abdelhadi, for his invaluable guidance, support, and encouragement throughout this research. His expertise and insights have been instrumental in shaping this work.

I am also grateful to Dr. Nicola Nicolici and Dr. Sorina Dumitrescu for their constructive feedback valuable suggestions and support that significantly enhanced the quality of this work.

Special thanks to my professors and colleagues at McMaster University for their support and for fostering an inspiring research environment. Lastly, I extend my heartfelt appreciation to my family for their unwavering support and encouragement.

Table of Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Notation

S	Set of states.
A	Set of actions.
$\mathbb{P}(s', r \mid s, a)$	Transition probability distribution; probability of transitioning to state s' and receiving reward r given current state s and action a.
$R(s, a)$	Reward function; expected reward received after taking action a in state s.
$\gamma \in [0, 1)$	Discount factor; determines the importance of future rewards.
$\pi(a \mid s)$	Policy; probability of taking action a in state s.
$V^{\pi}(s)$	State-value function; expected return starting from state s following policy π.

$Q^{\pi}(s, a)$	Action-value function; expected return starting from state s, taking action a, and thereafter following policy π.
G_t	Return; cumulative discounted reward from time t.
δ_t	Temporal-Difference (TD) error at time t.
$\lambda \in [0, 1]$	Parameter controlling the bias-variance trade-off in GAE.
θ	Policy parameters.
ϕ	Value function parameters.
$\hat{R}_t$	Estimated rewards-to-go.
$\hat{A}_t$	Estimated advantage function.
ϵ	Clipping parameter in PPO.
$r_t(\theta)$	Probability ratio between new and old policies.

Definitions

One-way standardization

Standardizing data and using it in its standardized form for computations, without reverting it to its original scale.

Two-way standardization

Standardizing data and then de-standardizing it back to its original scale before using it in computations.

Block Standardization (BS)

A method of standardizing data by aggregating data across multiple batches within defined blocks, standardizing the data based on the block's statistics.

Dynamic Standardization (DS)

A method of standardizing data where the standardization parameters (mean and standard deviation) are updated dynamically over time, using running statistics that account for all previously processed data.

Processing Element (PE)

A component in a hardware architecture designed for efficiently performing GAE computations, often forming part of a systolic array.

Abbreviations

AI	Artificial Intelligence
RL	Reinforcement Learning
PPO	Proximal Policy Optimization
GAE	Generalized Advantage Estimation
SoC	System-on-Chip
MDP	Markov Decision Process
DS	Dynamic Standardization
BS	Block Standardization
PE	Processing Element
BRAM	Block Random Access Memory
ReL	Rewards Loader
VaL	Values Loader
HEPPO	Hardware-Efficient Proximal Policy Optimization

Declaration of Academic Achievement

I, Hazem Taha, declare that this thesis titled, *HEPPO: Hardware-Efficient Proximal Policy Optimization. A Universal Pipelined Architecture for Generalized Advantage Estimation*, and the work presented in it are my own. I confirm that:

(i) This work was done wholly or mainly while in candidature for a research degree at McMaster University.

(ii) Where any part of this thesis has previously been submitted for a degree or any other qualification at this or any other institution, this has been clearly stated.

(iii) Where I have consulted the published work of others, this is always clearly attributed.

(iv) Where I have quoted from the work of others, the source is always given.

(v) This thesis is entirely my own work.

Chapter 1

Introduction

This chapter provides a comprehensive overview of Reinforcement Learning (RL), highlighting its significance and diverse applications across multiple domains. It identifies the primary challenges faced by RL algorithms, particularly in resource-constrained environments, and outlines the objectives and structure of this thesis aimed at addressing these challenges.

1.1 Overview of Reinforcement Learning

RL is a subset of machine learning in which an agent develops decision-making capabilities by engaging with an environment to achieve a certain goal. The agent systematically gathers observations and rewards from the environment and employs them to enhance its policy for the purpose of maximizing cumulative rewards over time [8]. The interaction loop is illustrated in Figure 1.1.

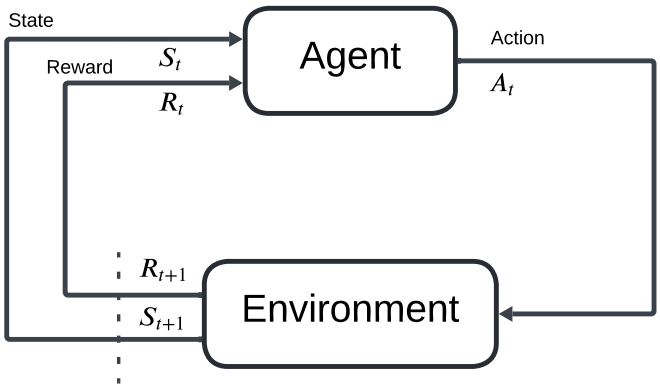

Figure 1.1: The Reinforcement Learning Framework. Original illustration based on Bhatt at al. [1].

The reinforcement learning paradigm has become a critical component of modern artificial intelligence applications, demonstrating substantial success in a variety of fields, such as robotics, autonomous driving, healthcare, finance, and gaming. In robotics, RL enables robots to learn and adapt to dynamic environments, executing complex tasks such as object manipulation and navigation [3, 18, 24]. Autonomous driving gains from RL by enhancing decision-making abilities, enabling cars to operate safely and efficiently in varied and complex environments [43]. In the healthcare sector, RL enhances patient outcomes by personalizing medicine and optimizing treatment plans to fit the unique requirements of each patient [3]. Furthermore, RL has transformed gaming by allowing artificial intelligence to excel at superhuman proficiency in games like chess and Go [40, 41].

In addition, RL has substantial applications in finance, industrial control, energy management, and telecommunication. In finance, RL-based trading bots learn and

adapt to market conditions, optimizing trading tactics in real-time [2, 50]. In industrial control, RL enhances operational efficiency in resource allocation and predictive maintenance by dynamically modifying control parameters to maximize system performance and minimize operating costs [7, 25]. In energy management, RL is employed to optimize energy usage and improve the efficiency of smart grids by forecasting and addressing energy demands [26]. In telecommunications, RL optimizes network traffic, improves service quality, and minimizes latency through dynamic bandwidth allocation and resource management [29].

Nevertheless, RL faces extreme obstacles, including the necessity for extensive data, substantial memory requirements, and high computational demands. These obstacles can restrict its practicality and scalability in real-world applications. Confronting these problems is crucial for progressing the field and facilitating wider acceptance of RL technologies.

1.2 Problem Definition and Motivation

Although RL algorithms exhibit considerable potential, their adoption in real-world, resource-limited settings presents a difficulty. Complex RL algorithms face significant challenges in being used on hardware platforms with limited resources, like embedded systems and edge devices, because they need a lot of memory and computing resources which causes a significant drop in their response time and performance [17, 44].

Proximal Policy Optimization (PPO) is a widely adopted RL algorithm for its robustness and effectiveness in handling complex tasks, demonstrating notable success in multi-agent applications such as logistics and supply chain management [49], as well as in industrial scenarios like job shop scheduling within manufacturing [4], due

to its adaptability and sample efficiency [39]. Nonetheless, PPO's computational limitations along with its considerable memory requirements, leave it a poor choice for hardware-constrained applications [44].

To bridge the gap between advanced RL algorithms such as PPO and practical hardware implementations, this thesis addresses the following challenges and sets corresponding objectives:

- **Computational Bottlenecks:** Identify and optimize the computationally intensive components of the PPO algorithm.
- **Memory Constraints:** Investigate PPO's memory requirements and implement efficient data representation and storage techniques.
- **Hardware-Efficient Algorithm Modifications:** Modify the PPO algorithm to ensure compatibility with hardware accelerators and generalizability without compromising learning efficacy.
- **Integration into System-on-Chip (SoC) Architectures:** Explore and design the seamless integration of the entire PPO pipeline onto a single SoC to enhance data throughput and reduce communication overhead.

Overall Goal: Enable the deployment of the PPO algorithm in resource-constrained environments by addressing computational and memory challenges, optimizing algorithmic efficiency, and ensuring seamless integration with hardware architectures, all without sacrificing performance.

1.3 Thesis Structure

This thesis is organized as follows:

- **Chapter 2: Background** — Background Offers an extensive summary of the essential concepts and techniques pertinent to this thesis, encompassing the principles of reinforcement learning, Generalized Advantage Estimation, and Proximal Policy Optimization.
- **Chapter 3: Literature Review** — Investigates the necessity for accelerating deep reinforcement algorithms such as PPO, analyzes previous studies about the optimization of PPO for hardware applications, and highlights deficiencies and gaps in present research which this study intends to rectify.
- **Chapter 4: Methodology** — Outlines the algorithmic modifications and the architecture of the accelerator accelerator. This chapter encompasses the suggested data standardization, quantization methodologies, and the architecture of the HEPPO system.
- **Chapter 5: Results and Discussion** — Displays the experimental findings assessing the efficacy of the proposed hardware and software methodologies. This chapter examines the efficacy of HEPPO in alleviating computational constraints and improving training efficiency.
- **Chapter 6: Conclusion and Future Work** — Summarizes the principal contributions of the thesis, examines the consequences of the findings, and delineates prospective avenues for future study to enhance hardware-accelerated reinforcement learning algorithms.

Chapter 2

Background

Building upon the introduction, this chapter provides a comprehensive overview of the fundamental concepts and algorithms in reinforcement learning essential for understanding the challenges addressed in this thesis. By delving into the core principles like value functions, policy optimization methods, specifically Proximal Policy Optimization (PPO), as well as advantage functions, we lay the groundwork for the subsequent exploration of hardware-efficient implementations.

2.1 Fundamentals of Reinforcement Learning

An RL problem is often formalized as a Markov Decision Process (MDP), characterized by the Markov property, which posits that the future state depends solely on the current state and action, not on the sequence of preceding events:

$$\mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a, \text{history}) = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.1.1}$$

An MDP is defined by the tuple $(S, A, \mathbb{P}, R, \gamma)$, as described in the Notation section. Where:

- S is the set of **states** representing possible configurations of the environment.
- A is the set of **actions** available to the agent at each state.
- $\mathbb{P}(s' \mid s, a)$ is the **Transition probability** defining the likelihood of moving from state s to s' after action a.
- $R(s, a)$ is the **Reward function** assigning a real-valued reward for each state-action pair.
- γ is the **Discount factor** $(0 \leq \gamma < 1)$ determining the importance of future rewards.

2.1.1 Finite Markov Decision Processes (FMDPs)

If the state and action spaces are finite, then the process is called a **Finite Markov Decision Process** (FMDP). FMDPs are particularly important to the theory of reinforcement learning as they constitute the majority of modern RL problems [42]. They are mathematically tractable and allow for the application of dynamic programming methods.

Figure 2.1 shows a useful way to summarize the dynamics of a finite MDP through a transition graph, illustrated using a simple recycling robot example.

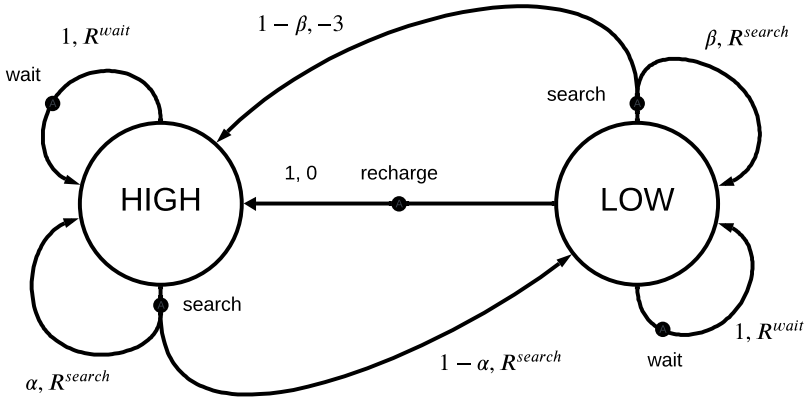

Figure 2.1: Finite MDP Transition Graph Example. The robot can be in one of two states: "Searching" or "Recharging". The actions available and the probabilities of transitioning between states are indicated. Original illustration based on Sutton at al. [42].

In this example, the robot operates in two states: *Searching* and *Recharging*. The robot can choose to search for recyclable cans or go back to recharge its battery. Each action has associated probabilities of transitioning to the next state and receiving certain rewards. This finite MDP allows us to model the decision-making process and optimize the robot's policy to maximize its expected cumulative reward.

2.1.2 Goal of Reinforcement Learning

The goal of reinforcement learning is to find a policy π that maximizes the expected cumulative reward, also known as the expected return. A policy $\pi(a|s)$ defines the probability of taking action a when in state s.

The return G_t is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1.2}$$

where $\gamma \in [0, 1)$ is the discount factor that determines the present value of future rewards, and R_{t+k+1} is the reward received at time $t + k + 1$.

2.1.3 Value Functions

To evaluate the quality of states and actions, RL utilizes value functions, which estimate the expected return starting from a state or state-action pair under a particular policy.

State-Value Function $V^\pi(s)$ The state-value function $V^\pi(s)$ is defined as the expected return when starting from state s and following policy π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s \right] \tag{2.1.3}$$

Action-Value Function $Q^\pi(s, a)$ The action-value function $Q^\pi(s, a)$ is defined as the expected return when starting from state s, taking action a, and thereafter following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s, A_t = a \right] \tag{2.1.4}$$

The optimal policy π^* maximizes the expected return for all states. The optimal

state-value function $V^*(s)$ and optimal action-value function $Q^*(s,a)$ are defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.1.5}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.1.6}$$

If the optimal action-value function $Q^*(s,a)$ is known, the optimal policy π^* can be obtained by selecting actions that maximize $Q^*(s,a)$:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{2.1.7}$$

2.2 Estimating the Action-Value Function

An essential aspect of RL algorithms is estimating the action-value function $Q^{\pi}(s,a)$ for the current policy π. Estimating $Q^{\pi}(s,a)$ is crucial for policy evaluation and improvement.

There are two primary methods for estimating $Q^{\pi}(s,a)$:

2.2.1 Monte Carlo Methods

Monte Carlo methods estimate $Q^{\pi}(s,a)$ by averaging the returns observed after visiting state s and taking action a over multiple episodes. The procedure is as follows:

- **Data Collection:** Run multiple episodes following policy π. For each occurrence of state s and action a, record the total discounted return G_t from that time step until the end of the episode.

- **Estimation:** Estimate $Q^\pi(s, a)$ as the average of these returns:

$$Q^\pi(s, a) \approx \frac{1}{N(s, a)} \sum_{i=1}^{N(s,a)} G_t^{(i)} \tag{2.2.1}$$

  where $N(s, a)$ is the number of times action a was taken in state s, and $G_t^{(i)}$ is the return observed in the i-th occurrence.

Monte Carlo methods have the following characteristics:

- **Advantages:**
  - Simple to implement.
  - Unbiased estimates in the limit of infinite samples.
- **Disadvantages:**
  - High variance due to the stochastic nature of returns.
  - Requires episodes to terminate (not suitable for continuous tasks without modifications).
  - Slow convergence in some cases.

2.2.2 Temporal-Difference Methods

Temporal-Difference (TD) methods estimate $Q^\pi(s, a)$ by bootstrapping from the current estimate of $Q^\pi(s', a')$. One common TD method is the n-step bootstrapping approach.

One-Step TD Learning

At each time step t:

- **Observe:** Immediate reward R_{t+1} and next state S_{t+1}.
- **Update Rule:**

$$Q^{\pi}(S_t, A_t) \leftarrow Q^{\pi}(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) - Q^{\pi}(S_t, A_t)\right] \quad (2.2.2)$$

  where:

  - α: Learning rate ($0 < \alpha \leq 1$).
  - A_{t+1}: Action taken in state S_{t+1} according to policy π.

n-Step TD Methods

TD methods can be extended to use rewards from the next n steps:

$$Q^{\pi}(S_t, A_t) \leftarrow Q^{\pi}(S_t, A_t) + \alpha \left[\sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^{n} Q^{\pi}(S_{t+n}, A_{t+n}) - Q^{\pi}(S_t, A_t)\right] \quad (2.2.3)$$

Characteristics of TD Methods

- **Advantages:**
  - Can learn from incomplete episodes (suitable for continuing tasks).

  - Typically lower variance than Monte Carlo methods.
  - Update estimates after every time step (online learning).

- **Disadvantages:**
  - Introduce bias due to bootstrapping.
  - Choice of n affects bias-variance trade-off.

2.2.3 Comparison Between Monte Carlo and TD Methods

The key differences between Monte Carlo and TD methods are:

- **Monte Carlo Methods:**
  - Use actual returns from complete episodes.
  - Unbiased but can have high variance.
  - Not suitable for non-terminating tasks without adjustments.
- **TD Methods:**
  - Use estimates of future returns (bootstrapping).
  - Introduce bias but typically have lower variance.
  - Suitable for online learning and continuing tasks.

2.3 Taxonomy of Reinforcement Learning Algorithms

Reinforcement Learning algorithms are broadly classified into **Model-Based** and **Model-Free** approaches [52]. Model-Free methods are further divided into **Value-Based**, **Policy-Based**, and **Actor-Critic** methods. This taxonomy aids in understanding the underlying mechanisms of different algorithms and their applicability to various problems.

2.3.1 Model-Based vs. Model-Free RL

Table 2.1: Comparison between Model-Based and Model-Free RL

Aspect	**Model-Based RL**	**Model-Free RL**
Environment Model	Utilizes a model of the environment to predict future states and rewards	Does not require a model; learns directly from interactions
Planning	Capable of planning by simulating future trajectories	Relies on trial-and-error learning
Sample Efficiency	Generally more sample-efficient due to planning	Often requires more samples to learn optimal policies
Complexity	Computationally intensive due to model learning and planning	Simpler to implement; lower computational overhead
Examples	Dyna-Q, Model Predictive Control	Q-Learning, SARSA, REINFORCE, PPO

2.3.2 Model-Free RL Algorithms

Model-Free algorithms learn directly from interactions with the environment without building an explicit model. They are categorized into Value-Based, Policy-Based, and Actor-Critic methods.

Table 2.2: Summary of Model-Free RL Algorithms

Category	**Algorithms**	**Key Characteristics**
Value-Based	Q-Learning, SARSA, DQN	Learn value functions $V(s)$ or $Q(s,a)$ to derive policies; suitable for discrete action spaces; may struggle with high-dimensional state spaces
Policy-Based	REINFORCE, Policy Gradient Methods, TRPO	Directly learn policy $\pi_\theta(a\|s)$; effective in continuous action spaces; can handle stochastic policies
Actor-Critic	A3C, DDPG, PPO	Combine actor (policy) and critic (value function); leverage advantages of both value-based and policy-based methods; suitable for complex environments

The key characteristics of these methods are:

- **Value-Based Methods:** Focus on estimating value functions to indirectly derive optimal policies. They are suitable for problems with discrete action

spaces but may face challenges in high-dimensional state spaces.

- **Policy-Based Methods:** Optimize the policy directly by adjusting parameters to maximize expected rewards. They are effective in environments with large or continuous action spaces and can handle stochastic policies.

- **Actor-Critic Methods:** Incorporate both policy and value function estimation, combining the strengths of both approaches for stability and applicability to complex tasks.

2.3.3 Actor-Critic Methods

Actor-Critic methods combine policy-based and value-based approaches to leverage the advantages of both [16, 21]. They consist of two main components:

- **Actor:** Learns the policy $\pi_\theta(a|s)$ responsible for selecting actions. The actor updates the policy parameters θ to improve the policy.

- **Critic:** Estimates the value function $V^\pi(s)$ or the action-value function $Q^\pi(s, a)$, using parameters ϕ. The critic evaluates the actions taken by the actor and provides feedback in the form of an advantage estimate, which guides the actor's updates.

The interaction between the actor and critic is depicted in Figure 2.2. The actor selects actions based on the current policy, and the critic evaluates these actions to update the value function. This collaboration helps in reducing variance in policy gradient updates and accelerates learning [20].

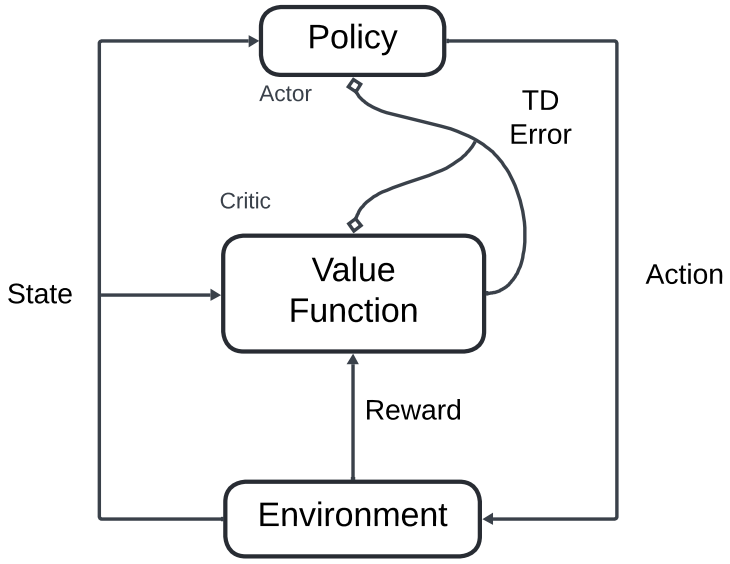

Figure 2.2: Actor-Critic Architecture. The actor selects actions based on the policy, while the critic evaluates these actions to update the value function. Original illustration based on Sutton at al [42].

Policy and Value Function Updates

In Actor-Critic Methods, the actor and the critic are typically modeled using **deep neural networks** as function approximators [20, 38, 39]. This allows the framework to handle complex, high-dimensional state and action spaces effectively. Consequently, the updates to the parameters θ and ϕ correspond to neural network parameter adjustments performed through gradient-based optimization techniques.

Actor Update. The **actor network** parameterized by θ represents the policy $\pi_\theta(a|s)$. To improve the policy, the actor performs **gradient ascent** on the expected return. This process iteratively adjusts θ to increase the probability of actions that

yield higher advantages and vice versa.

$$\theta \leftarrow \theta + \alpha \cdot \hat{A}_t(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{2.3.1}$$

Where:

- α is the learning rate for the actor network.
- $\hat{A}_t(s_t, a_t)$ is the estimated advantage at time step t, indicating the relative value of action a_t in state s_t.
- $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is the gradient of the log-probability of the taken action with respect to the actor's parameters.

This update rule is applied iteratively during training, allowing the actor to progressively refine the policy towards optimality by favoring actions that lead to higher expected returns.

Critic Update. The **critic network** parameterized by ϕ estimates the value function $V^\pi(s; \phi)$. The critic updates its parameters by minimizing the **mean squared error** between the predicted value and a target value, utilizing **gradient descent**.

$$\phi \leftarrow \phi - \beta \cdot \nabla_\phi \left(V^\pi(s_t; \phi) - \text{target}\right)^2 \tag{2.3.2}$$

Where:

- β is the learning rate for the critic network.

- The *target* is defined as:

$$\text{target} = \begin{cases} r_{t+1} + \gamma V^{\pi}(s_{t+1}; \phi) & \text{(Temporal-Difference)} \\ G_t & \text{(Monte Carlo)} \end{cases} \tag{2.3.3}$$

  where r_{t+1} is the reward received after taking action a_t in state s_t, γ is the discount factor, and G_t is the cumulative return from time step t.

The critic's update is also applied iteratively during training to reduce the prediction error, thereby providing more accurate value estimates that the actor relies on to improve the policy.

Over time, this collaborative optimization leads to the convergence of the actor towards an optimal policy π^* and the critic towards an accurate value function $V^{\pi^*}(s)$, ultimately enhancing the agent's decision-making capabilities.

Advantage Function

The advantage function $\hat{A}_t(s_t, a_t)$ quantifies how much better or worse an action a_t is compared to the expected value for state s_t under policy π. It is defined as:

$$\hat{A}_t(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{2.3.4}$$

Depending on how $Q^{\pi}(s_t, a_t)$ is estimated, the advantage can be computed in different ways:

Temporal-Difference (TD) Approach. In the TD approach (one-step bootstrapped), the advantage is estimated using the temporal-difference residual δ_t^V, which

measures the discrepancy between the predicted value and the actual reward received plus the estimated value of the next state:

$$\delta_t^V = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{2.3.5}$$

Thus, the advantage estimate becomes:

$$\hat{A}_t(s_t, a_t) = \delta_t^V \tag{2.3.6}$$

Monte Carlo Approach. In the Monte Carlo approach, the advantage is estimated using the rewards-to-go, which sums all future discounted rewards starting from time t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3.7}$$

The advantage estimate is then:

$$\hat{A}_t(s_t, a_t) = G_t - V^\pi(s_t) \tag{2.3.8}$$

2.3.4 Generalized Advantage Estimation (GAE)

Generalized Advantage Estimation (GAE) extends the advantage function by providing a flexible mechanism to balance the trade-off between bias and variance in advantage estimation [37]. GAE achieves this by aggregating multi-step temporal-difference residuals, resulting in more stable and efficient learning.

Temporal-Difference Residual

As previously defined, the temporal-difference residual δ_t^V is:

$$\delta_t^V = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{2.3.9}$$

k-Step Advantage Estimate

Building upon the TD residual, the k-step advantage estimate $\hat{A}_t^{(k)}$ aggregates k consecutive TD residuals, each discounted by γ:

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) - V(s_t) \tag{2.3.10}$$

This formulation incorporates rewards over k steps and a bootstrapped estimate of the value function at step $t + k$, effectively blending multi-step returns with a baseline to estimate the advantage.

Generalized Advantage Estimation

GAE extends the k-step advantage estimates by introducing a weighting parameter $\lambda \in [0, 1]$, which controls the trade-off between bias and variance. The GAE $\hat{A}_t^{\text{GAE}}$ is defined as an exponentially weighted sum of the k-step advantage estimates:

$$\hat{A}_t^{\text{GAE}} = (1-\lambda)\sum_{k=1}^{\infty}\lambda^{k-1}\hat{A}_t^{(k)} \tag{2.3.11}$$

$$= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V \tag{2.3.12}$$

This can be efficiently computed using the recursive formulation:

$$\hat{A}_t = \delta_t^V + \gamma\lambda\hat{A}_{t+1} \tag{2.3.13}$$

with the recursion starting from $\hat{A}_T = \delta_T^V$ at the terminal time step T.

Bias-Variance Trade-Off

The parameter λ controls the balance between bias and variance in the advantage estimates:

- $\lambda = 1$: GAE reduces to the Monte Carlo estimation, which has low bias but high variance.
- $\lambda = 0$: GAE reduces to the one-step TD error, which has high bias but low variance.
- Intermediate values of λ blend multi-step returns, providing a balance between bias and variance.

2.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy-gradient method that improves training stability through a clipped surrogate objective function, preventing large policy updates [39]. PPO addresses issues of high variance and instability in policy updates by limiting the change in the policy at each update [39, 53].

Both the policy (actor) and the value function (critic) in PPO are typically modeled using deep neural networks as was explained in subsection 2.3.3. The parameters of these networks are updated using gradient-based optimization methods, facilitating the learning of optimal policies and accurate value estimates.

The PPO algorithm as presented in algorithm 1 comprises three main components: trajectory collection, advantage and rewards-to-go calculation, and loss calculation with backpropagation.

Trajectory Collection

In this phase, the agent interacts with the environment using the current policy π_{θ_k} to collect a set of trajectories $D_k = \{\tau_i\}$ by inferencing using the actor and critic neural networks. Each trajectory consists of states, actions (from the actor), value estimates (from the critic), and rewards observed during an episode.

Advantage and Rewards-to-Go Calculation

After collecting trajectories, the rewards-to-go $\hat{R}_t$ are computed for each time step t. Subsequently, the advantage estimates $\hat{A}_t$ are calculated using Generalized Advantage Estimation (GAE), which leverages the current value function V_{ϕ_k} as defined in

Section 2.3.4. These estimates provide a measure of how much better an action is compared to the expected value, guiding the policy updates.

Loss Calculation and Backpropagation

The core of PPO involves updating both the policy and value networks using loss functions that incorporate gradient-based optimization. Specifically:

- **Policy Update**: The policy network (actor) with parameters θ is updated by maximizing the PPO-Clip objective function. This objective ensures that the new policy does not deviate excessively from the old policy π_{θ_k}, thereby maintaining training stability.
- **Value Function Update**: The value network (critic) with parameters ϕ is updated by minimizing the mean squared error between the predicted values $V_\phi(s_t)$ and the computed rewards-to-go $\hat{R}_t$.

Both updates utilize backpropagation and stochastic gradient descent (or its variants) to adjust the neural network parameters based on the gradients of the respective loss functions. Although not explicitly shown in algorithm 1, it is common in PPO to perform multiple network updates within the same training epoch to enhance learning efficiency.

Input: Initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect a set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment;

Compute rewards-to-go $\hat{R}_t$;

Compute advantage estimates $\hat{A}_t$ using GAE;

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

Update the value function by minimizing the squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2$$

end

Algorithm 1: PPO-Clip Variant by OpenAI [33]

2.4.1 Advantages of PPO

PPO offers several benefits:

- **Stability**: The clipping mechanism prevents destructive policy updates.
- **Sample Efficiency**: By reusing collected data for multiple epochs of optimization.
- **Simplicity**: PPO is relatively easy to implement and tune compared to other advanced policy optimization methods.

2.5 Summary

This chapter laid out the foundational concepts of reinforcement learning, covering MDPs, value functions, and key algorithms like PPO. Understanding these principles is crucial for appreciating the challenges in hardware-efficient implementations of RL algorithms. In the next chapter, the existing literature on hardware acceleration techniques for reinforcement learning will be explored, identifying gaps that our research aims to fill.

Chapter 3

Literature Review

This chapter provides a comprehensive literature review of the computational demands of deep reinforcement learning and the need for custom hardware acceleration. The objective is to systematically narrow down the search scope in order to identify the gaps and challenges that drive the development of hardware-efficient solutions for deep RL algorithms like Proximal Policy Optimization (PPO) through the examination of existing research.

3.1 Computational Demands and Hardware Limitations in Reinforcement Learning

3.1.1 Escalating Computational Requirements

Modern reinforcement learning (RL) algorithms rely mainly on deep neural networks as the primary function approximators for agent policies [20, 38, 39]. This section examines the escalating computational demands associated with these core deep neural

networks.

The growth of data and model sizes in deep learning has resulted in a huge increase in computational demands, with resource requirements scaling quadratically or more in relation to model complexity [45]. The computational power needed by deep learning models has markedly increased, substantially exceeding the growth of general-purpose hardware performance, as depicted in Figure 3.1.

Commodity computing like CPUs is becoming ever less capable of satisfying these growing demands. This deficiency is worsened by the slowdown of Moore's Law, which has traditionally forecasted the doubling of transistors on integrated circuits roughly every two years. The deceleration of Moore's Law has markedly limited the ability of CPUs to meet the increasing requirements of deep learning tasks [45].

The increasing processing demands present considerable difficulties in effectively facilitating large-scale deep learning applications with general-purpose hardware. As a result, a major percentage of the enhancement in performance has been achieved by executing models for extended durations on a greater number of machines, a strategy that is both economically unappealing and less energy-efficient [45].

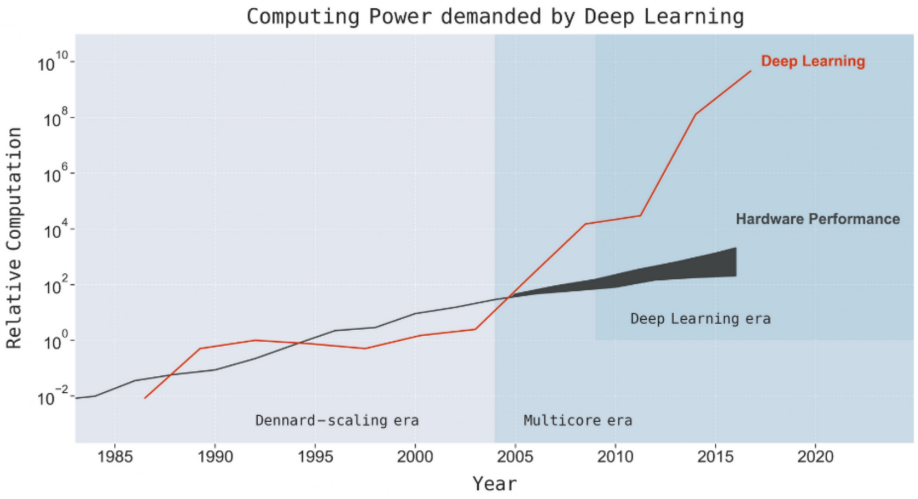

Figure 3.1: Computing power used in deep learning models compared with hardware performance growth. Adapted from Thompson et al. (2022) with permission [45].

3.1.2 Necessity for Specialized Hardware

Specialized hardware solutions like graphics processing units (GPUs) and tensor processing units (TPUs) are key to overcoming the limits of general-purpose CPUs and fulfilling the increasing computing requirements of deep learning. They are engineered to accommodate the parallel processing demands intrinsic to deep learning tasks and provide a substantial performance enhancement compared to conventional CPUs.

Compared to their initial implementations, GPUs have exhibited substantial performance advances, with a 35× increase in speed by 2012 [45]. Their architecture, which enables massive parallelism, is the primary reason for this remarkable acceleration. This architecture is well-suited for tasks such as matrix multiplications and other operations that are prevalent in deep learning algorithms.

Additionally, TPUs have made substantial strides in computational efficiency, achieving a 4.9× increase in compute per watt between 2017 and 2020 [45]. They are

specifically engineered to increase the efficiency and performance of tensor operations, the core of deep learning models.

With these developments have been made, GPUs and TPUs are still not universally applicable, particularly in situations where there are strict real-time, area, power, and resource constraints. In applications that require minimal latency, constrained physical space, or reduced power consumption, the size and power requirements of GPUs and TPUs make them less feasible. Alternate solutions or additional specialized hardware may be required in these situations to satisfy the distinctive requirements without sacrificing performance [45].

Therefore, while the utilization of GPUs and TPUs is useful in optimizing the computational requirements of modern deep learning models, their constraints in limited situations underscore the need for better-customized hardware design to accommodate a wider array of applications.

3.2 Impact of Computational Constraints on Reinforcement Learning Performance

3.2.1 Inference Time and Real-Time Performance

Timely decision-making is important in real-world applications, and computational delays can have a substantial impact on performance. Deep reinforcement learning models frequently exhibit inference durations of several hundred milliseconds, which can be deleterious in real-time contexts when the environment keeps in changing while the agent determines the subsequent action [44]. In time-critical tasks such as autonomous driving or robotic control, delays in action selection may result in

suboptimal or perhaps disastrous consequences.

Linear models such as Augmented Random Search (ARS) [30] demonstrate significantly faster inference times compared to more computationally intensive deep learning models. ARS can perform inference in the order of milliseconds which makes it a better fit for real-time applications [44]. In contrast, algorithms like Probabilistic Ensembles with Trajectory Sampling (PETS), which rely on simulation-based planning [4], have substantially longer inference times, rendering them impractical for time-critical tasks.

Delays as small as 0.005 seconds can reduce performance by up to 40% in environments like Hopper-v2 and Humanoid-v2 [44]. This shows the importance of accounting for computational costs and inference times when deploying reinforcement learning algorithms in real-world scenarios, as illustrated in Figure 3.2.

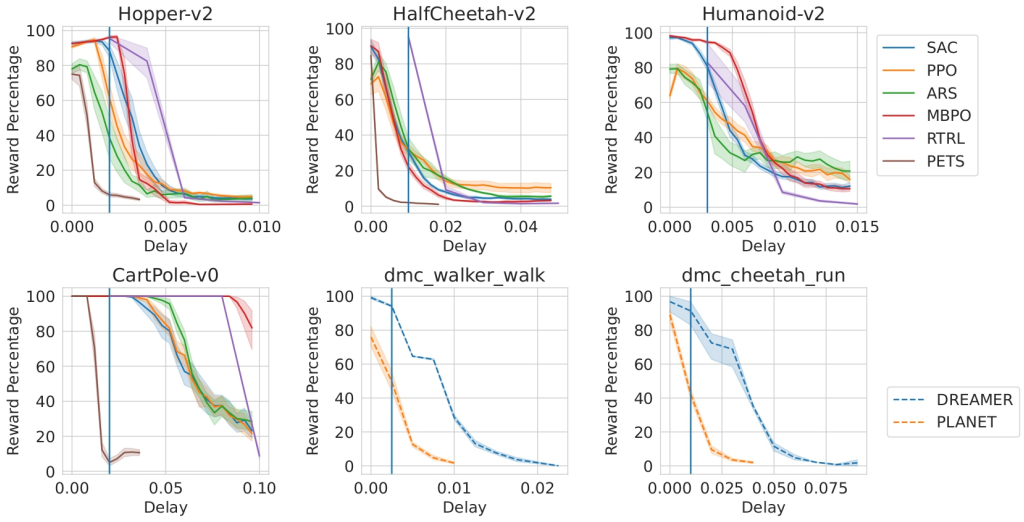

Figure 3.2: Effect of inference delay on reinforcement learning performance. Adapted from Thodoroff et al. (2022) with permission [44]. Degradation of performance on continuous control environments with varying amounts of inference delay. The vertical blue bar represents one timestep during training.

3.2.2 Performance Under Limited Compute Resources

Resource constraints further worsen the impact of computational delays on reinforcement learning performance. In scenarios where computational resources are limited, such as embedded systems or edge devices, the choice of algorithm becomes critical. Thodoroff et al. [44] demonstrated that with less than 20% of available compute power, simple linear models like ARS outperform more complex deep reinforcement learning algorithms like Proximal Policy Optimization (PPO) [39], Soft Actor-Critic (SAC) [20], and Model-Based Policy Optimization (MBPO) [22]. This is primarily due to the lower inference times of linear models under constrained computational budgets. As computational resources increase, deep reinforcement learning algorithms begin to leverage the additional compute power to improve decision-making and performance. However, this comes at the cost of higher inference times, which may not be acceptable in real-time applications with strict latency requirements. Figure 3.3 illustrates the performance degradation of various algorithms under limited computational resources.

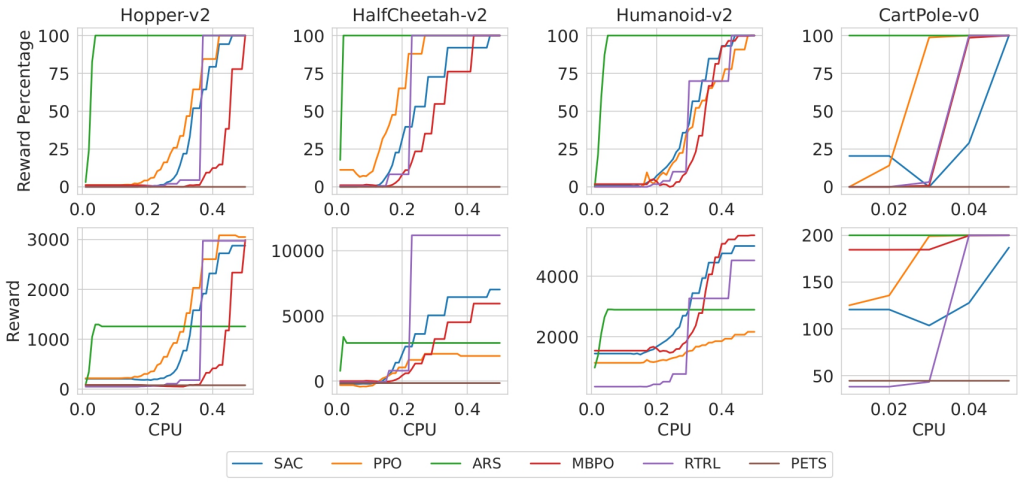

Figure 3.3: Performance adjusted by the inference delay on varying amounts of CPU. Adapted from Thodoroff et al. (2022) with permission [44]. The top row shows the performance degradation when varying the amount of CPU. The bottom rows display the reward obtained by each algorithm.

3.2.3 Robustness of Reinforcement Learning Algorithms to Inference Delays

The robustness of reinforcement learning algorithms to inference delays varies significantly among different methods. Thodoroff et al. evaluated several algorithms under varying amounts of computational delay and found notable differences [44]. Specifically, Real-Time Reinforcement Learning (RTRL) exhibits greater resilience to inference delays due to its design, which accounts for action delays during training by incorporating the previous action as part of the input state. This approach allows RTRL to anticipate and compensate for delays, maintaining performance in real-time settings.

Model-based algorithms, such as Model-Based Policy Optimization (MBPO), show improved robustness to delays compared to model-free methods. By utilizing a model

of the environment to predict future states, these algorithms can produce more informed actions even when actions are delayed, thereby mitigating the negative impact of inference latency.

In contrast, simulation-based algorithms like PETS suffer significantly from inference delays due to their substantial computational overhead. The intensive computations required for simulation-based planning result in longer inference times making these algorithms impractical for time-critical applications where rapid decision-making is needed.

These findings shows that accounting for computational delays during the training phase can enhance the robustness of reinforcement learning algorithms in real-time settings.

3.3 Necessity of Specialized Hardware for Efficient Reinforcement Learning Training

3.3.1 Intensive Compute Resources and Training Time

Modern reinforcement learning applications like AlphaGo Zero highlight the immense computational efforts needed. AlphaGo Zero required approximately 40 days of training, during which it played 29 million self-play games [41]. This immense computational effort underscores the intensive resource demands of reinforcement learning algorithms, especially when training models to surpass human-level performance.

The use of Tensor Processing Units (TPUs) contributed heavily to the overall success of AlphaGo Zero, significantly reducing computation time. AlphaGo Zero used

four TPUs on a single machine and achieved faster and more efficient training compared to AlphaGo Lee, which required 48 TPUs distributed across multiple machines [41]. This demonstrates the critical need for hardware acceleration in reinforcement learning to handle intensive training workloads and reduce time-to-convergence.

3.3.2 Computational Load Despite Algorithmic Advances

Even with newer, more efficient algorithms, the computational demands remain substantial. In the Procgen benchmark, training deep reinforcement learning agents required up to 200 million steps across 500 levels [17]. The experiments conducted by Govindarajan et al. consumed more than 10,000 GPU hours on 32 GPUs. Each training run for a single game took approximately 24 hours, illustrating the intensive need for high-performance hardware. Despite algorithmic advances like the Phasic Policy Gradient (PPG) method [5], the computational demands remain similar to those of previous algorithms like PPO. This indicates that improving algorithms alone does not drastically reduce the need for large computational resources.

3.4 Computational Challenges of Proximal Policy Optimization

3.4.1 Significance and Widespread Adoption of PPO

Proximal Policy Optimization (PPO) has established itself as a cornerstone algorithm in deep reinforcement learning due to its robustness, sample efficiency, and ease of implementation [39]. Balancing performance with computational tractability, PPO

has been widely adopted in both academic research and practical applications across various domains.

In multi-agent environments, PPO has demonstrated remarkable capabilities. Yu et al. [49] found that PPO-based methods achieve surprisingly strong performance in cooperative multi-agent games, often matching or surpassing state-of-the-art off-policy algorithms in terms of final returns and sample efficiency. This highlights PPO's adaptability and effectiveness in complex settings where multiple agents interact and learn concurrently.

In industrial applications, PPO has been employed to tackle complex optimization problems. Chen et al. [4] utilized an LSTM-PPO-based reinforcement learning algorithm to address the dynamic job shop scheduling problem (JSP). Traditional scheduling methods, such as heuristic rules and mathematical programming, often struggle with the complexity and dynamic nature of modern manufacturing systems. The LSTM-PPO algorithm dynamically adjusts scheduling strategies in response to changes in the production environment, leading to optimal scheduling decisions. Their experimental results showed that the LSTM-PPO algorithm outperformed other methods, including DQN-based RL algorithms and traditional heuristic rules, in terms of convergence speed and scheduling efficiency.

These successes underscore PPO's versatility and effectiveness, reinforcing its significance in the field of reinforcement learning.

3.4.2 Computational Demands of PPO

Despite its advantages, applying PPO in complex environments and with limited computational resources introduces significant challenges, some of which were introduced in Section 3.2. PPO involves iterative optimization of policy and value networks, requiring extensive interaction with the environment to collect data, followed by computationally intensive gradient updates. The need to balance exploration and exploitation, along with the stability constraints imposed by the proximal policy updates, results in high computational overhead.

In multi-agent settings, the computational load is amplified as PPO must process observations and actions of multiple agents simultaneously [49]. This increases the dimensionality of the input and output spaces, leading to larger neural networks and longer training times. The simultaneous learning and coordination among agents add layers of complexity, demanding more from computational resources.

In the dynamic job shop scheduling problem, the LSTM-PPO algorithm must handle various combinations of job instructions, machine states, and operation sequences [4]. This results in a complex and high-dimensional state space that poses challenges for RL training and exploration. Training recurrent neural networks like LSTM adds further computational complexity, requiring substantial resources and time, especially when high-quality data is necessary for effective learning.

Moreover, the need for timely decision-making in real-world applications imposes strict requirements on inference time. Delays in action selection can negatively impact performance, particularly in time-critical tasks. As discussed earlier, even small delays can lead to significant performance degradation [44].

3.5 Accelerating PPO and Environment Simulations

Efficient training of reinforcement learning algorithms like PPO necessitates addressing computational bottlenecks in both algorithmic computations and environment simulations. Despite advancements in algorithmic efficiency, the computational demands remain substantial, requiring specialized hardware and optimization techniques to accelerate training and inference.

3.5.1 Hardware Acceleration of PPO

Meng et al. [32] proposed a high-throughput accelerator for PPO on CPU-FPGA platforms to tackle the computational challenges inherent in PPO's neural network operations. PPO relies on neural network computations during both the inference and training phases, which can be computationally intensive and present considerable bottlenecks, especially in resource-constrained environments.

To address these challenges, the authors developed an accelerator that leverages a systolic-array-based architecture to enable parallel processing of neural network operations. This architecture enables efficient matrix multiplications—fundamental to neural network computations—by systematically organizing data flow and computation units to maximize data reuse and throughput. The systolic array facilitates the pipelined execution of operations, significantly reducing computation time.

An overview of the FPGA accelerator is shown in Figure 3.4. The accelerator deploys two Compute Units (CUs), one for the value network and one for the policy

network. Each CU is composed of a 2D systolic array to perform forward propagation (FW), backward propagation (BW), gradient computation, and weight updates (WU). The detailed architecture of a single CU is depicted in Figure 3.5. It includes an activation module, an activation derivative module, and buffers to store network weights and intermediate results, such as the weights buffer for network weights, the z^l buffer for immediate matrix products, and the a^l and δ^l buffers for FW activation results and BW errors, respectively.

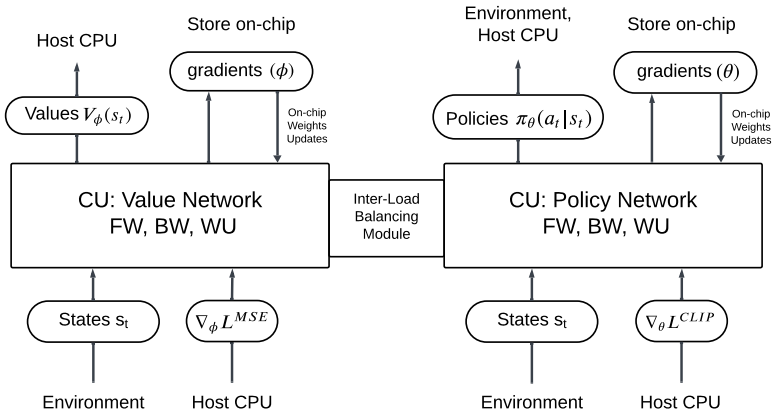

Figure 3.4: High-level overview of the FPGA accelerator. Original illustration based on Meng et al. (2020) [32].

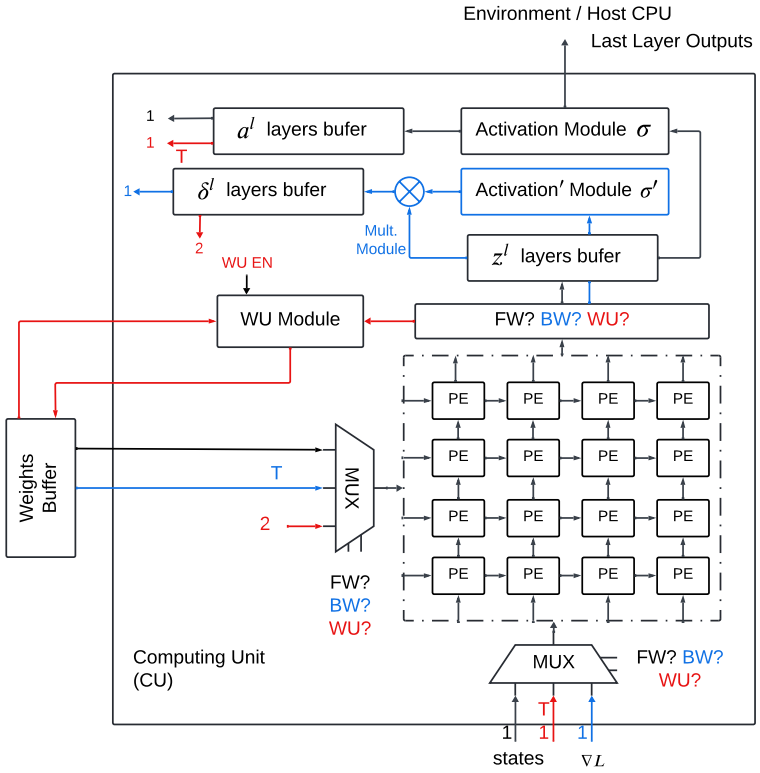

Figure 3.5: Detailed architecture of a single Compute Unit (CU). Original illustration based on Meng et al. (2020) [32].

Furthermore, Meng et al. optimized data access patterns by implementing a memory-blocked data layout. This approach organizes data in memory to minimize access conflicts and improve cache utilization, improving overall computational efficiency. By partitioning data into blocks that align with the architecture of the systolic array, they reduced the overhead associated with memory access, which is critical for high-performance computing.

To address the imbalance in computational loads between the value network and the policy network—two critical components of PPO—the authors introduced a compute sharing technique. This technique dynamically allocates computational

resources between the two networks based on demand, ensuring balanced utilization and preventing resource underutilization. By sharing computation units, the accelerator efficiently handles varying workloads, maximizing hardware utilization.

The accelerator was rigorously evaluated using the Hopper and Humanoid environments from OpenAI Gym, making substantial throughput improvements—up to a 30.5-fold increase over CPU-only implementations and up to a 27.5-fold increase over CPU-GPU implementations. These gains underscore the effectiveness of specialized hardware acceleration in improving the performance of PPO.

While these results are impressive, several limitations remain. The accelerator primarily focuses on accelerating neural network computations and does not address the computational overhead associated with other phases of the algorithm like The Generalized Advantage Estimation (GAE) and the environment simulations. In reinforcement learning training, each of the mentioned phases can consume a substantial portion of the processing time, up to nearly half of the total time in some environments, as shown in Table 4.2. Therefore, the overall training speedup is constrained by the unaddressed bottleneck in environment simulation.

Additionally, the accelerator's reliance on storing neural network parameters and intermediate results entirely on-chip imposes scalability constraints. FPGAs have limited on-chip memory, which restricts the size of neural networks that can be accommodated. This limitation raises difficulties when scaling to larger networks required for more complex tasks or environments with higher-dimensional state and action spaces.

Moreover, the design is tailored specifically for PPO implementations that use

Multilayer Perceptron (MLP) architectures. This specialization limits the generalizability of the accelerator to other reinforcement learning algorithms or network architectures, such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs), commonly used in various applications.

Lastly, the development and deployment of FPGA-based accelerators require specialized hardware design expertise. Implementing such accelerators involves hardware description languages and tools that may not be familiar to all practitioners in the reinforcement learning community, potentially limiting the widespread adoption of this approach.

3.5.2 Environment Acceleration Techniques

Efficiently simulating environments is a key element in successful training complex reinforcement learning algorithms like the PPO. This is because interactions with these environments usually constitute a major part of the computational burden, as shown in Table 4.2. A variety of methods have been developed to improve the performance of reinforcement learning environments by leveraging both CPU and GPU architectures.

For CPU-based acceleration, Weng et al. [47] introduced EnvPool: a high-performance engine for environment execution designed to tackle the inefficiencies linked to slow environment simulations. EnvPool make use of a C++ thread pool-based executor engine along with Python wrappers which allow it to support both synchronous and asynchronous execution styles. This structural design enables EnvPool to efficiently handle a large number of environment instances in parallel and hence maximizes the potential of multi-core CPU architectures.

On a NVIDIA DGX-A100 system equipped with 256 CPU cores, EnvPool achieved remarkable performance reaching up to 1 million frames per second (FPS) for Atari environments and 3 million FPS for MuJoCo environments. Moreover, EnvPool exhibited great improvements even on regular laptops where it achieved a 14.9-fold increase for Atari and a 19.2-fold increase for MuJoCo compared to standard Python implementations. These findings shows the efficacy of optimizing environment execution on CPU architectures, particularly when specialized hardware such as GPUs is not available.

Advances in GPU-based acceleration techniques have also greatly improved the performance of environment simulations. One significant contribution is CuLE (CUDA Learning Environment) by Dalton et al. [6]. CuLE is a framework for reinforcement learning that utilizes GPU parallelism to speed up the simulation of Atari games. Running thousands of environment instances simultaneously on a single GPU, CuLE effectively exploits the substantial parallel processing capabilities found in modern GPUs.

A major innovation of CuLE is its capability to render frames directly on the GPU, thereby avoiding the CPU-GPU communication bottleneck that is often seen in GPU-accelerated applications. This strategic design choice not only minimizes latency but also enhances simulation throughput. CuLE has achieved great performance, hitting rates of up to 155 million frames per hour on a single GPU—performance levels that were previously only attainable through extensive CPU clusters.

From a practical standpoint, the implementation of CuLE significantly reduced the training time for the Atari Pong game, decreasing it from 21.2 minutes with the traditional OpenAI Gym environment to just 5.9 minutes through a more efficient

batching strategy. In addition, CuLE attained training frame rates ranging from 26,000 to 68,000 frames per second (FPS) on a single GPU, with rates increasing to 187,000 FPS when distributed across four GPUs. These results underscore the potential of GPU-based acceleration to dramatically improve the efficiency of reinforcement learning training, especially in environments that can take full advantage of the data parallelism provided by GPU architectures.

Nonetheless, GPU-based acceleration techniques, including CuLE, face several challenges. The Single Instruction, Multiple Data (SIMD) architecture of GPUs may induce thread divergence, which occurs when threads within a warp follow different execution paths due to conditional branching. This divergence can lead to performance issues, as threads that deviate cannot execute in parallel. CuLE seeks to mitigate this challenge through careful kernel design and structured environmental simulations aimed at reducing divergence; however, this remains an inherent limitation of GPU architectures.

Moreover, there is a significant demand for ample GPU memory to concurrently emulate thousands of environment instances. In contrast to CPUs, GPUs have limited memory capacities, which can be quickly depleted during large-scale environment simulations. This limitation can constrain the scalability of GPU-based methods, particularly in environments that are memory-intensive or when trying to increase the number of simultaneous instances.

Furthermore, CuLE has been specifically designed for Atari games, which are characterized by simpler dynamics and graphical needs. Adapting this approach for more intricate environments—especially those involving complex simulations or graphics—might require substantial alterations or may be impractical due to hardware

limitations.

In the field of robotics, Liang et al. [27] created a reinforcement learning simulator that is GPU-accelerated, utilizing NVIDIA Flex, a physics engine designed for GPUs. Their research is concentrated on sophisticated locomotion challenges in robotics, which require heavy computational physics simulations. By harnessing the power of GPUs to parallelize simulations involving hundreds to thousands of robots, the authors significantly enhanced training efficiency.

They successfully trained a humanoid running task in under 20 minutes using a single GPU and CPU core, processed 60,000 simulation frames per second with 750 humanoid simulations, and showed considerable scaling advantages in experiments involving multiple GPUs. Although these GPU-accelerated simulators provide remarkable performance improvements, they also present certain drawbacks. The necessity for advanced GPU hardware may not be feasible for every user, especially those in environments with limited resources. Furthermore, these simulators are typically tailored for particular types of tasks or environments, which constrains their use in other areas.

3.6 Research Gaps in Custom Hardware for Reinforcement Learning

Despite significant advancements in accelerating neural network computations and environment simulations, there remains a notable gap in the development of custom hardware solutions tailored specifically for complex deep reinforcement learning algorithms like Proximal Policy Optimization (PPO). While hardware accelerators have

been designed for various deep RL algorithms—predominantly value-based methods such as Deep Q-Networks (DQNs)—the focus on policy optimization algorithms like PPO has been limited.

PPO is widely recognized for its robustness, sample efficiency, and effectiveness across a range of reinforcement learning applications [39, 49]. However, the computational demands of PPO, especially when deployed in real-world or resource-constrained environments, present significant challenges that have not been adequately addressed by existing hardware accelerators. Current research efforts have primarily concentrated on accelerating neural network computations within the PPO [32], often neglecting other computationally intensive components of the algorithm, such as the generalized advantage estimation and the environment interactions.

This gap becomes clearer the more the specific application is constrained. This includes applications with low power consumption, real-time performance requirements, and limited computing and memory resources. These factors, which are essential for the deployment of RL algorithms in embedded systems and edge devices, are frequently disregarded by current hardware accelerators. Closing this gap requires comprehensive understanding of PPO's computational characteristics, identifying bottlenecks such as memory access patterns and opportunities for parallelization and optimization.

Recognizing these research gaps is essential for advancing the field of RL. Developing specialized hardware for PPO can enhance its efficiency and extend its applicability to a broader range of real-world applications where computational constraints are a significant concern. This thesis aims to address these challenges by exploring custom hardware architectures for PPO and investigating algorithmic modifications

to improve performance in resource-constrained settings. By targeting the unaddressed bottlenecks in the PPO pipeline, it aims to contribute to the development of more efficient and scalable reinforcement learning systems suitable for real-world, resource-limited environments.

3.7 Summary

This chapter examined the increasing computational requirements of reinforcement learning and the constraints of existing hardware solutions. It emphasized the importance of specialized hardware in order to satisfy the performance requirements of modern RL algorithms and identified the deficiencies in current research, particularly in the context of PPO. The findings highlight the need of creating hardware-efficient methods to facilitate the actual implementation of reinforcement learning in resource-limited settings. Chapter 4 will detail the proposed algorithmic adjustments and hardware solutions aimed at mitigating these problems and improving the efficiency of PPO implementations.

Chapter 4

Methodology

This chapter details the methodology used to address the computational constraints of hardware-limited applications for optimizing the Proximal Policy Optimization (PPO) algorithm. The study begins with a thorough time profiling analysis to identify key bottlenecks within PPO's execution that can benefit from hardware acceleration. Profiling PPO's core phases across diverse OpenAI Gym environments on both CPU-GPU and CPU-only systems, insights into the computationally intensive components were gathered to help target optimization efforts effectively. Following this analysis, the design of a custom hardware accelerator—HEPPO (Hardware-Efficient Proximal Policy Optimization)—is introduced to address the demands of Generalized Advantage Estimation (GAE) computation. It leverages parallel processing and massive pipelining to enhance data throughput and minimize latency. PPO algorithmic modifications were further implemented to ensure compatibility with the hardware architecture and improve PPO's efficiency and stability in resource-constrained settings. This includes data standardization, quantization, and optimized memory management techniques. Finally, the chapter explores the integration of the entire PPO

pipeline onto a System-on-Chip (SoC) platform to consolidate all different phases onto a single chip, greatly reducing latency, enhancing real-time performance, and lowering power consumption.

4.1 Time Profiling and Bottleneck Identification

Optimizing the efficiency of the PPO algorithm for constrained applications requires a thorough understanding of its computational bottlenecks. A detailed time profiling analysis was conducted to identify the most time-consuming components that could benefit from optimization or hardware acceleration.

4.1.1 Profiling Methodology

To identify computational bottlenecks within the Proximal Policy Optimization (PPO) algorithm, time profiling was conducted across four diverse OpenAI Gym environments: BipedalWalker-v3 [10], CartPole-v1 [11], HalfCheetah-v5 [12] and Humanoid-v5 [13]. These environments were chosen to represent a wide range of complexities and dynamics, ensuring that the identified bottlenecks are reflective of PPO's performance across various domains and minimizing potential biases in the results.

Descriptions of the Environments

- **HalfCheetah-v5**: A two-dimensional simulation of a bipedal robot resembling a cheetah. The agent controls six joints to make the robot run as fast as possible in the forward direction. The action space is continuous with six dimensions, each bounded between -1 and 1, representing torques applied at the hinge

joints [12]. The observation space consists of 17 continuous variables, including positions and velocities of the robot's body parts, excluding the robot's x-coordinate by default.

- **Humanoid-v5**: A high-dimensional control task involving a three-dimensional humanoid robot with 17 hinge joints. The agent must learn to walk forward without falling. The action space is 17-dimensional, continuous, and bounded between -0.4 and 0.4, representing torques applied at the joints. The observation space includes 376 variables capturing detailed state information such as joint positions and velocities [13]. The complexity arises from the high dimensionality and the need for balance and coordination.

- **CartPole-v1**: A classic control problem where a pole is attached by an unactuated joint to a cart that moves along a frictionless track. The agent applies a force to the cart by choosing one of two discrete actions: push the cart to the left (action 0) or push the cart to the right (action 1), aiming to prevent the pole from falling over. The action space is discrete with two possible actions. The observation space is four-dimensional, consisting of the cart position, cart velocity, pole angle, and pole angular velocity [11].

- **BipedalWalker-v3**: A challenging task where the agent controls a two-legged robot to walk across rough terrain. The action space is continuous with four dimensions, corresponding to motor speed values applied at the hip and knee joints of each leg, ranging from -1 to 1. The observation space comprises 24 continuous variables, including hull angle, angular velocities, horizontal and vertical speeds, joint angles, joint angular speeds, contact sensors, and 10 lidar

rangefinder measurements [10]. The environment tests the agent's ability to balance and adapt to uneven surfaces.

- **LunarLander-v3**: A classic rocket trajectory optimization problem where the agent controls a lunar lander to safely descend and land on the moon's surface. The action space is discrete with four possible actions: do nothing, fire left orientation engine, fire main engine, or fire right orientation engine. The observation space is eight-dimensional, consisting of the lander's position, velocity, angle, angular velocity, and Boolean flags indicating whether each leg is in contact with the ground [14]. The environment challenges the agent's ability to control the lander's descent and landing.

PPO Configuration

Neural Network: A standardized neural network architecture was used for both the actor and critic in time profiling and subsequent experiments on quantization and standardization, serving as a static benchmark to isolate the effects of these optimizations. The architecture consisted of a feedforward neural network with three fully connected layers, each containing 64 neurons and utilizing ReLU activation functions. This design strikes a balance between computational efficiency and sufficient capacity to approximate the necessary value functions and policies [39].

Hyperparameters: Key hyperparameters used during training are summarized in Table 4.1. These hyperparameters were selected based on standard practices and adjusted through preliminary experimentation to ensure stable learning across all environments.

Table 4.1: Hyperparameters used for PPO Training

Hyperparameter	**Value**	**Description**
Number of updates per iteration	5	Number of gradient updates to the actor and critic networks per training iteration.
Learning rate (α)	3×10^{-4}	Learning rate for the optimizer.
Learning rate decay	1×10^{-5}	Exponential decay rate for the learning rate schedule.
Discount factor (γ)	0.95	Discount factor for future rewards in the return calculation.
Clipping parameter (ϵ)	0.2	Clipping parameter for the probability ratio to prevent large policy updates.
GAE parameter (λ)	0.98	Smoothing parameter for Generalized Advantage Estimation (GAE).

System Description

Profiling was conducted on two systems to observe the impact of hardware configurations:

- **CPU-GPU System**: A 32-core Intel Xeon Silver 4216 CPU @ 2.10 GHz with an NVIDIA Tesla V100-SXM2-32GB GPU.
- **CPU-Only System**: The same CPU without GPU acceleration.

Key Measurements

For each environment and system, the execution time of key phases within the PPO algorithm was measured across hundreds of iterations to ensure statistical significance.

The profiled phases included:

1. **DNN Inference**: The forward pass of the neural network to predict actions given states.
2. **Environment Run**: Execution of the environment's dynamics to generate trajectories based on the agent's actions.
3. **GAE Computation**: Calculation of the Generalized Advantage Estimation using the collected trajectories.
4. **Network Updates**: Computation of loss and backpropagation to update the neural network parameters.

The percentage of total execution time spent in each phase was calculated to identify the primary computational bottlenecks.

4.1.2 Profiling Results

The profiling results are summarized in Table 4.2 and Table 4.3 and visualized in Figure 4.1 and Figure 4.2. These results illustrate the distribution of execution time across different phases of the PPO algorithm for each environment on both systems.

Table 4.2: Time Profiling of PPO Iteration (CPU-GPU System) across Four Environments

Component	**HalfCheetah (%)**	**Humanoid (%)**	**CartPole (%)**	**BipedalWalker (%)**
DNN Inference	8.0	9.9	7.0	6.4
Environment Simulation	36.8	46.6	23.7	47.3
CPU-GPU Communication	0.7	0.8	0.9	0.5
Storing Trajectories	10.2	5.7	12.8	8.6
GAE Memory Fetch	7.5	5.0	10.1	6.0
GAE Computation	26.8	24.8	37.6	22.2
GAE Memory Write	0.6	0.2	0.9	0.5
Loss Calculation	6.8	5.2	2.8	6.2
Backpropagation	2.6	1.8	4.3	2.3

Table 4.3: Time Profiling of PPO Iteration (CPU-Only System) across Four Environments

Component	**HalfCheetah (%)**	**Humanoid (%)**	**CartPole (%)**	**BipedalWalker (%)**
DNN Inference	8.0	10.5	8.4	7.4
Environment Simulation	25.9	60.7	37.3	32.6
Storing Trajectories	7.4	4.7	8.0	6.8
GAE Memory Fetch	6.2	3.5	7.0	5.4
GAE Computation	21.7	11.2	25.5	19.4
GAE Memory Write	0.5	0.3	0.7	0.4
Loss Calculation	25.3	6.1	4.8	23.4
Backpropagation	5.0	2.9	8.4	4.6

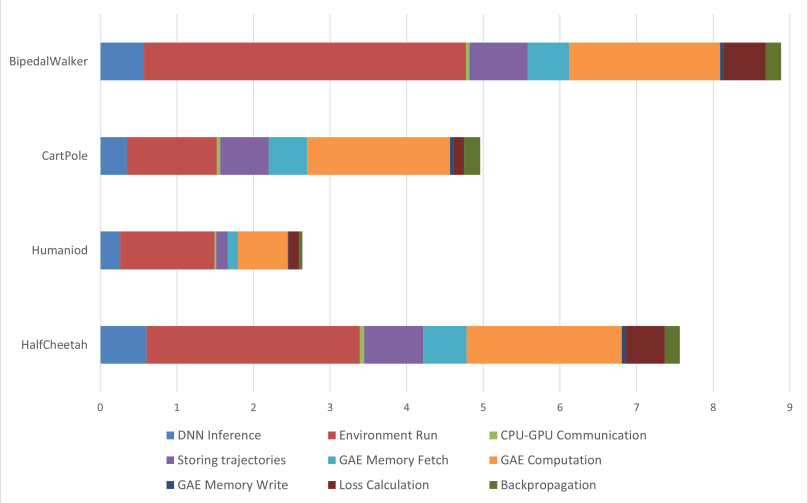

Figure 4.1: Time Profiling of PPO Iteration on CPU-GPU System across Four Environments

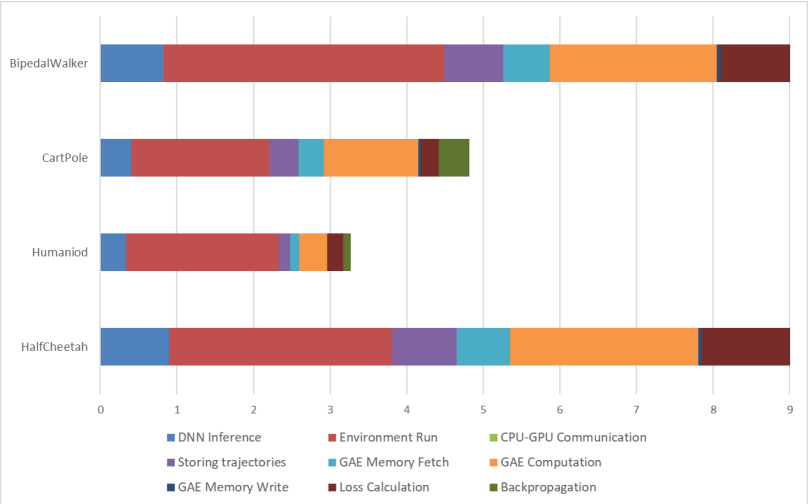

Figure 4.2: Time Profiling of PPO Iteration on CPU-Only System across Four Environments

4.1.3 Analysis of Bottlenecks

The profiling results reveal that **Environment Simulation** and **Generalized Advantage Estimation (GAE)** computation are the primary bottlenecks in the PPO

algorithm across all tested environments and system configurations.

In the Environment Simulation phase, execution time varies depending on the complexity of the environment. As on the CPU-only system, it consumes a minimum of **25.9%** (HalfCheetah-v5) and a maximum of **60.7%** (Humanoid-v5) of the total execution time. On the CPU-GPU system, it ranges from **23.7%** (CartPole-v1) to **47.3%** (BipedalWalker-v3). This substantial time consumption is attributed to the computational demands of simulating environment dynamics, especially in tasks with high-dimensional state spaces and complex physics like Humanoid-v5.

Despite its large time consumption, accelerating environment simulation was not the focus of this research. This is because environment simulations vary from one application to another, not a characteristic of the PPO, making it an impractical target to develop a generalized accelerator. These simulations are often implemented in high-level languages and accelerated through CPU-based and GPU-based techniques discussed in the literature, making implementing a custom hardware accelerator less feasible within the scope of this work.

The GAE computation phase was found as well to exhibit a significant computational overhead. On the CPU-GPU system, it ranges from **22.2%** (BipedalWalker-v3) to **37.6%** (CartPole-v1). Notably, in simpler environments like CartPole-v1, GAE computation constitutes a higher percentage of the total execution time due to shorter environment simulation times, making the relative overhead of GAE more pronounced. Moreover, accounting for the GAE data transfers from and to the main memory, this renders the overall GAE phase on the CPU-GPU system to range from **28.7%** (BipedalWalker-v3) to **48.6%** (CartPole-v1). This is quite similar to the range consumed by the Environment Simulation phase.

The persistent challenge of the entire GAE phase as a bottleneck irrespective of environmental complexity or system configuration emphasizes its significant impact on the efficiency of PPO training. Unlike environment simulation, GAE is an intrinsic component of the PPO algorithm, making it a universal target for optimization.

Other phases, such as backpropagation and deep neural network (DNN) inference for trajectory collection, were found to consume a smaller fraction of the total execution time. On the CPU-only system, DNN Inference ranges from **7.4%** (BipedalWalker-v3) to **10.5%** (Humanoid-v5), while on the CPU-GPU system, it ranges from **6.4%** (BipedalWalker-v3) to **9.9%** (Humanoid-v5). Similarly, loss calculation has a minimal impact on overall execution time on the CPU-GPU system. Notably, loss calculation steps inherently involve using the critic network to compute new value estimates, highlighting the effectiveness of GPU acceleration in reducing the time consumed during this step.

While GPU acceleration effectively reduces the time for DNN inference and loss calculation, it does not significantly alleviate the computational burden of GAE computation. In some cases, offloading GAE computations to the GPU may introduce overheads that negate the benefits, particularly in simpler environments where data transfer and kernel launch times become relatively significant.

CPU-GPU communication accounts for less than **1%** of the total execution time across all environments on the CPU-GPU system, indicating that data transfer overhead is minimal and not a primary bottleneck.

Therefore, focusing on accelerating the GAE phase, including GAE computation and data transfer, presents a universally applicable opportunity to enhance PPO

training efficiency across a wide range of tasks and hardware configurations. By optimizing this critical phase—which accounts for up to **48.6%** of execution time—the research aims to improve the scalability and applicability of PPO in resource-constrained and real-time applications.

4.2 Proposed Solution and Hardware Architecture

A specialized hardware accelerator named **HEPPO** (Hardware-Efficient Proximal Policy Optimization) is proposed to address the identified bottleneck in GAE computation. HEPPO aims to optimize both computational and memory efficiency, enabling concurrent processing of multiple trajectories and fully utilizing parallel processing units.

4.2.1 Overview of the Hardware Accelerator

The primary objective of HEPPO is to parallelize the GAE computation by processing multiple trajectories simultaneously, thereby minimizing delays caused by serial computation and reverse iteration inherent in traditional implementations. By integrating specialized hardware components and optimizing memory management, HEPPO enhances data throughput and reduces communication overhead, achieving a worst-case time complexity of $O(T)$ instead of $O(T \times N)$, where T is the number of timesteps and N is the number of agents. Each timestep is processed significantly faster in HEPPO than in traditional methods due to this parallelism.

4.2.2 HEPPO Micro-Architecture Components

HEPPO's micro-architecture comprises several specialized components designed for efficient data handling and computation, as illustrated in Figure 4.3. The main components are the **Rewards Loaders (ReLs)**, responsible for loading elements from the rewards vector for each timestep; the **Values Loaders (VaLs)**, which fetch corresponding value estimates for each reward element; and the **Processing Elements (PEs)**, organized in a one-dimensional systolic array where each row processes distinct vectors from different agents or trajectories.

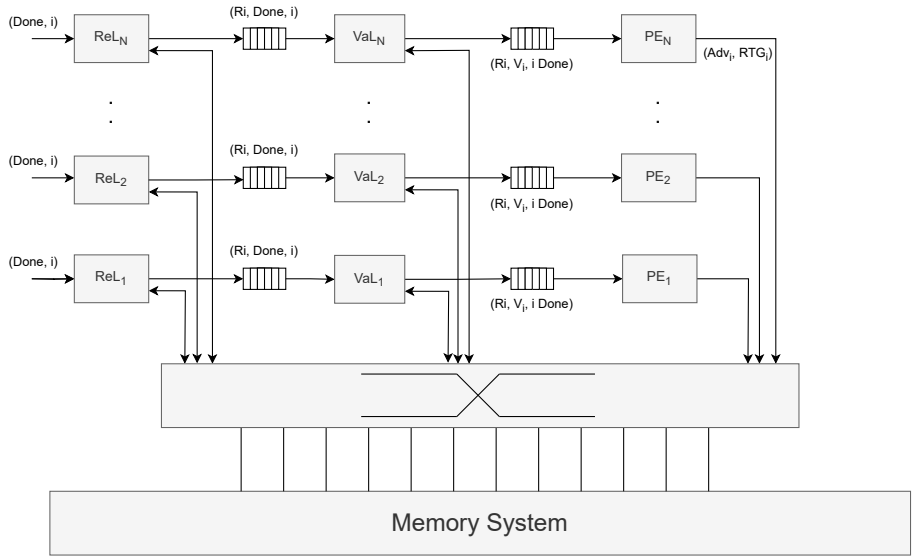

Figure 4.3: HEPPO Micro-Architecture Overview

The architecture leverages parallelism by assigning different trajectories to separate rows in the systolic array. The number of rows corresponds to the number of agents or parallel environments. Each row operates concurrently and independently, allowing for simultaneous processing of multiple trajectories. Vectors are assigned to rows in a round-robin manner, optimizing resource utilization and throughput.

4.2.3 Processing Element (PE) Architecture

Each Processing Element (PE) is designed to perform the computations required for GAE efficiently. The initial PE architecture, depicted in Figure 4.4, breaks down the computation into several stages that can be heavily pipelined to achieve the desired operating frequency.

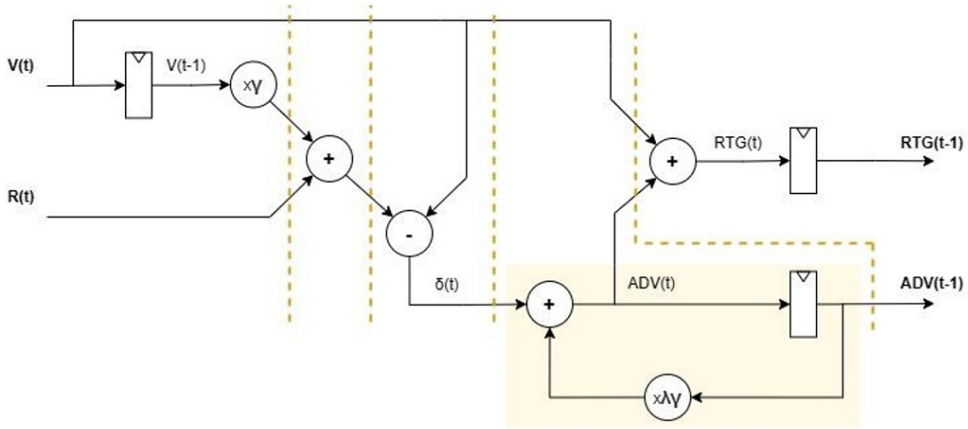

Figure 4.4: Initial PE Architecture with Potential Pipelining Stages

A critical challenge in pipelining the PE arises from the feedback loop inherent in the recursive nature of GAE computation. This feedback loop introduces data dependencies between consecutive timesteps, creating a bottleneck that limits the achievable frequency to approximately 275 MHz, far below the capabilities of advanced FPGAs, which can achieve frequencies exceeding 500 MHz in optimized designs. Attempting to pipeline the loop results in stalls, as later stages must wait for earlier computations to complete. To overcome this bottleneck, the computation is restructured to allow for effective pipelining while preserving the correctness of the GAE calculation.

4.2.4 k-Step Lookahead Solution

To address the feedback loop bottleneck in the GAE computation, a k-step lookahead approach is adopted, inspired by techniques used in pipelining recursive filters, particularly in IIR (Infinite Impulse Response) filter designs. These techniques are well-documented in Parhi's book on VLSI digital signal processing systems [34] and in various works on pipelined recursive filter implementations [28, 35, 36]. By reformulating the GAE computation, natural delays are introduced to align with pipeline stages, enabling efficient pipelining of the Processing Elements (PEs).

Mathematical Reformulation

The original GAE computation is defined as:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l} \tag{4.2.1}$$

where $\delta_t = r_t + \gamma V_{t+1} - V_t$ and $C = \gamma\lambda$.

By decomposing the advantage estimates at different timesteps, $\hat{A}_t$ can be expressed using future advantage estimates and δ terms. Table 4.4 illustrates this decomposition.

Table 4.4: Decomposition of Advantage Estimates for Different t Values

t	$\hat{\mathbf{A}}_\mathbf{t}$
T	$\hat{A}_T = \delta_T$
$T-1$	$\hat{A}_{T-1} = C\delta_T + \delta_{T-1}$ $= C\hat{A}_T + \delta_{T-1}$
$T-2$	$\hat{A}_{T-2} = C^2\delta_T + C\delta_{T-1} + \delta_{T-2}$ $= C^2\hat{A}_T + C\delta_{T-1} + \delta_{T-2}$ $= C\hat{A}_{T-1} + \delta_{T-2}$
$T-3$	$\hat{A}_{T-3} = C^3\delta_T + C^2\delta_{T-1} + C\delta_{T-2} + \delta_{T-3}$ $= C^3\hat{A}_T + C^2\delta_{T-1} + C\delta_{T-2} + \delta_{T-3}$ $= C^2\hat{A}_{T-1} + C\delta_{T-2} + \delta_{T-3}$ $= C\hat{A}_{T-2} + \delta_{T-3}$

From the table, it is evident that $\hat{A}_{T-3}$ can be expressed in terms of $\hat{A}_T$, $\hat{A}_{T-1}$, or $\hat{A}_{T-2}$, depending on the chosen formulation.

$$\begin{aligned}\hat{A}_{T-3} &= f_1(\delta_T, \delta_{T-1}, \delta_{T-2}, \delta_{T-3}) \\ &= f_2(\hat{A}_T, \delta_{T-1}, \delta_{T-2}, \delta_{T-3}) \\ &= f_3(\hat{A}_{T-1}, \delta_{T-2}, \delta_{T-3}) \\ &= f_4(\hat{A}_{T-2}, \delta_{T-3})\end{aligned} \tag{4.2.2}$$

In general, for calculating $\hat{A}_t$ with a generic k-step lookahead:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda,k)} = C^k \hat{A}_{t+k} + \sum_{l=0}^{k-1} C^l \delta_{t+l} \tag{4.2.3}$$

This reformulation allows the computation of $\hat{A}_t$ based on $\hat{A}_{t+k}$, introducing a natural delay of k timesteps. By computing $k-1$ additional δ terms and delaying the necessary values, the computation can be redesigned to fit a pipelined architecture without stalls.

Pipeline Integration

By adopting the k-step lookahead, the PE can be redesigned to include pipeline stages that align with the computation of $\hat{A}_t$ using delayed advantage estimates. Figure 4.5 illustrates the architecture of a PE utilizing a 3-step lookahead.

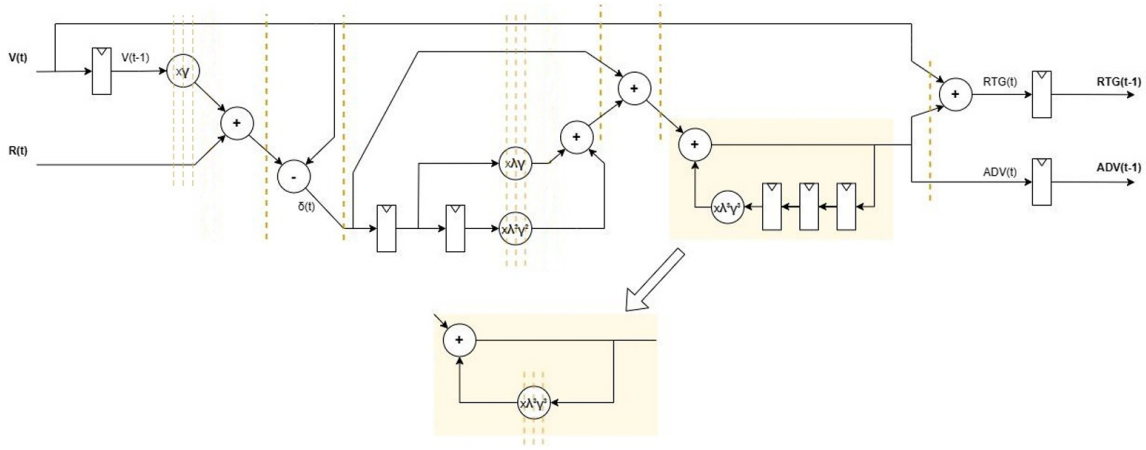

Figure 4.5: PE Architecture with 3-Step Lookahead Pipelining

In this architecture, the computation is broken down into stages that can be executed concurrently. The feedback loop is effectively unrolled over k steps, allowing the pipeline to operate without stalls. The necessary delays are incorporated naturally

into the pipeline stages, enabling the PE to achieve the maximum operating frequency and improve overall throughput.

4.2.5 Memory Bandwidth Bottleneck and On-Chip Stack BRAM Solution

Processing multiple trajectories in parallel introduces a significant demand on memory bandwidth. For a large-scale reinforcement learning setup with 64 agents and 1024 timesteps, the bandwidth required to keep all PEs fully utilized is substantial.

Memory Bandwidth Calculations

Assuming a single-precision floating-point format (32 bits) for rewards and value estimates, the required memory bandwidth can be calculated as follows. Each PE needs to read the reward and value for each timestep. For 64 agents:

$$\text{Read Bandwidth} = 64 \text{ agents} \times 2 \times 32 \text{ bits} = 512 \text{ bytes per clock cycle} \quad (4.2.4)$$

Similarly, each PE writes back the computed advantage and reward-to-go:

$$\text{Write Bandwidth} = 64 \text{ agents} \times 2 \times 32 \text{ bits} = 512 \text{ bytes per clock cycle} \quad (4.2.5)$$

The total bandwidth per clock cycle is:

$$\text{Total Bandwidth} = 1,024 \text{ bytes per clock cycle} \tag{4.2.6}$$

At an operating frequency of 300 MHz:

$$\text{Bandwidth per Second} = 1,024 \text{ bytes} \times 300 \times 10^6 = 307.2 \text{ GB/s} \tag{4.2.7}$$

This required bandwidth of approximately 307.2 GB/s far exceeds the capabilities of standard off-chip memory solutions like DDR4 3200 DRAM, which typically offer around 25 GB/s. This significant shortfall leads to underutilization of the PEs due to memory bandwidth limitations.

On-Chip Stack BRAM Solution

To address the memory bandwidth bottleneck and the data transfer latency associated with GAE Memory Fetch and GAE Memory Write that can reach up to 11% of the total training time in an environment like CartPole, an on-chip dual-port Block RAM (BRAM) solution is proposed. Utilizing BRAM leverages its high bandwidth and low latency, essential for sustaining the data flow required by the PEs and minimizing data transfer delays.

As shown in Figure 4.6, data is stored in a First-In Last-Out (FILO) structure, aligning with the reverse iterative access pattern of GAE computation. Rewards $R_{(i,t)}$ and value estimates $V_{(i,t)}$ of the same timestep (t) and across all agent trajectories (i) are stored together, enabling efficient simultaneous access. Here, T represents the

maximum number of timesteps, and N is the total number of agents. The use of dual-port BRAM allows concurrent read and write operations: while rewards and value estimates are read for the current timestep, computed advantages and rewards-to-go from a previous timestep are written back.

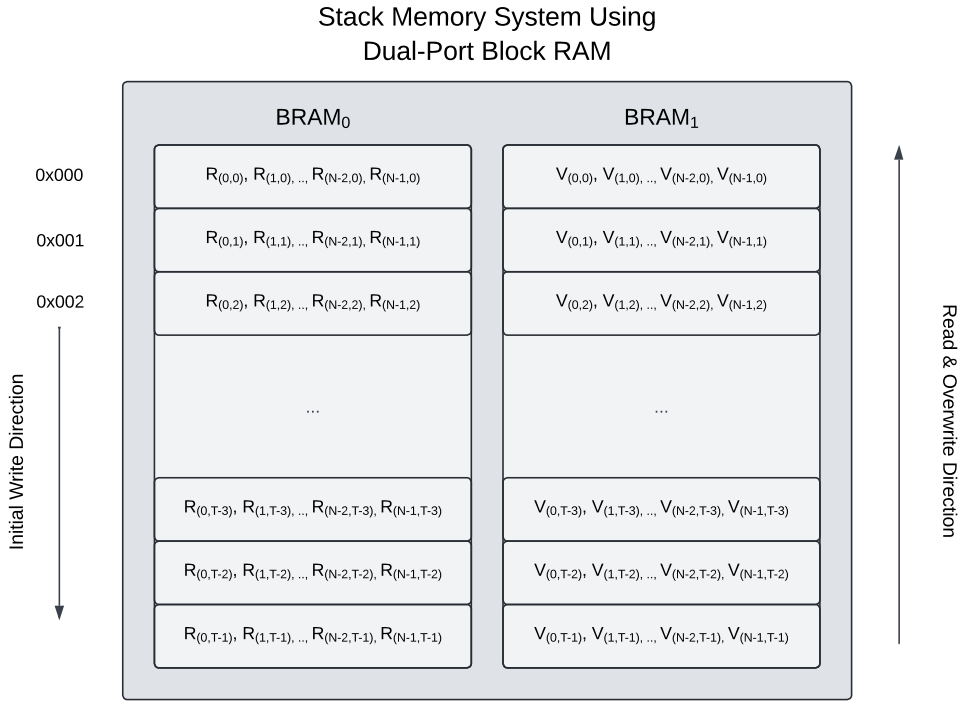

Figure 4.6: Dual-Port BRAM Stack Memory System

By moving data storage and processing on-chip, the stack BRAM memory system significantly reduces data transfer latency, achieving a single-clock latency for data movements. This effectively eliminates the overhead associated with GAE Memory Fetch and GAE Memory Write. Since data movements occur within the chip, the high-speed internal connections ensure minimal latency compared to off-chip memory accesses.

This memory architecture ensures that both the high bandwidth demands and

the low-latency requirements are met, allowing the PEs to operate at maximum efficiency. The ability to overwrite the same memory locations reduces memory usage and simplifies data management.

Although modern FPGAs provide sufficient BRAM capacity to accommodate the required memory, the bandwidth limitations of the on-chip memory still present a bottleneck. For instance, the total memory required is calculated as follows: for 64 agents and 1024 timesteps, storing rewards and value estimates requires $64 \times 1024 \times 2 = 131,072$ words. Assuming each word is 32 bits, the total memory requirement is $131,072 \times 32 = 4,194,304$ bits, or 4 Mb. While FPGAs like Xilinx UltraScale+ offer sufficient BRAM capacity, the bandwidth to read and write 32-bit data at the required rates is still constrained by the number of available BRAM ports and their operating frequency.

To further alleviate the bandwidth bottleneck and fully exploit the parallelism of the architecture, data rescaling is proposed in Section 4.4. This reduces the data width multiple folds, effectively decreasing the required memory bandwidth allowing for more efficient data transfer within the system and making the implementation more feasible on available hardware.

4.2.6 Data Layout and Memory Management

Efficient data layout is crucial for maximizing memory bandwidth and ensuring seamless data flow between memory and computation units. Algorithm 2 is implemented to manage the FILO stack structure in BRAM, ensuring efficient data access patterns compatible with the hardware architecture.

GAE Stack Memory Algorithm

The following algorithm outlines the memory layout and processing steps. Different memory blocks are utilized, such as the Rewards Memory Block (RMB), Value Estimates Memory Block (VEMB), Advantage Memory Block (AMB), and Rewards-to-Go Memory Block (RTGMB).

Input: Rewards $r[i][t]$, Values $V[i][t]$ for agents $i = 1$ to N, timesteps $t = 1$ to T

Output: Advantages $\hat{A}[i][t]$, Rewards-to-Go $\hat{R}[i][t]$

Initialize memory blocks RMB, VEMB, AMB, RTGMB;

`// Data Loading Phase`

for $t = 1$ *to* T **do**

  for $i = 1$ *to* N **do**

    Store $r[i][t]$ into RMB$[t][i]$;

    Store $V[i][t]$ into VEMB$[t][i]$;

  end

end

`// GAE Computation Phase`

`// Base Case: Compute` $\hat{A}_T$ `and` $\hat{R}_T$ `for all agents`

for $i = 1$ *to* N **do in parallel**

  Retrieve $r_T \leftarrow$ RMB$[T][i]$;

  Retrieve $V_T \leftarrow$ VEMB$[T][i]$;

  Compute $\hat{A}_T, \hat{R}_T$;

end

Algorithm 2: GAE Memory Layout and Processing

```
// Iteratively Compute Â_t and R̂_t while storing Â_{t+1} and R̂_{t+1}
for t = T − 1 to 1 do
    for i = 1 to N do in parallel
        Store Â_{t+1} into AMB[t + 1][i];
        Store R̂_{t+1} into RTGMB[t + 1][i];
        Retrieve r_t ← RMB[t][i];
        Retrieve V_t ← VEMB[t][i];
        Compute Â_t, R̂_t;
    end
end
```

Algorithm 2: GAE Memory Layout and Processing (Continued)

In the data loading phase, rewards and value estimates for all agents and timesteps are loaded into the BRAM memory blocks RMB and VEMB. The GAE computation phase proceeds backward in time, from timestep $T-1$ down to 1 with T being the base case. At each timestep iteration, previously calculated advantage estimates and rewards-to-go are being stored while current rewards and value estimates are being fetched from the BRAM and processed using the PEs. The inner loop over agents is executed in parallel, leveraging the parallel PEs in the hardware architecture. Computed advantages and rewards-to-go are stored back into the BRAM memory blocks AMB and RTGMB.

4.2.7 Integration with the PPO Algorithm

To fully exploit the benefits of HEPPO, the PPO algorithm is adjusted to align with the hardware architecture's data flow and computation patterns as will be detailed in Section 4.4. The algorithm processes data in batches corresponding to the number of agents and timesteps compatible with the hardware design. Data is quantized before being loaded into the BRAM, reducing the memory footprint and alleviating bandwidth constraints. The data flow is structured to match the FILO access pattern and the pipelined computation stages, ensuring efficient data movement and minimal latency.

4.3 Design Space Exploration of Full PPO Integration on SoC

Building upon the HEPPO microarchitecture designed to accelerate the Generalized Advantage Estimation (GAE) computation phase, this section explores the potential of integrating the entire Proximal Policy Optimization (PPO) pipeline into a single System-on-Chip (SoC). By consolidating all components of the PPO algorithm onto a single chip, latency and communication overhead can be minimized, thereby enhancing performance and enabling real-time reinforcement learning applications.

4.3.1 SoC Architecture Overview

The proposed architecture leverages the AMD-Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit, which integrates a quad-core Arm Cortex-A53 processing system (PS)

and a dual-core Arm Cortex-R5 real-time processor. This platform provides the necessary computational resources for running environment simulations and managing the high-level control flow of the PPO algorithm. The programmable logic (PL) section of the device offers extensive resources for custom logic implementation, including neural network inference and GAE computation.

Figure 4.7 illustrates the integration of the entire PPO pipeline within the SoC. The PS executes the environment simulations locally on the ARM Cortex-A53 cores, while the PL contains custom hardware accelerators, including the HEPPO accelerator for GAE computation and a neural network accelerator adapted from existing high-performance designs.

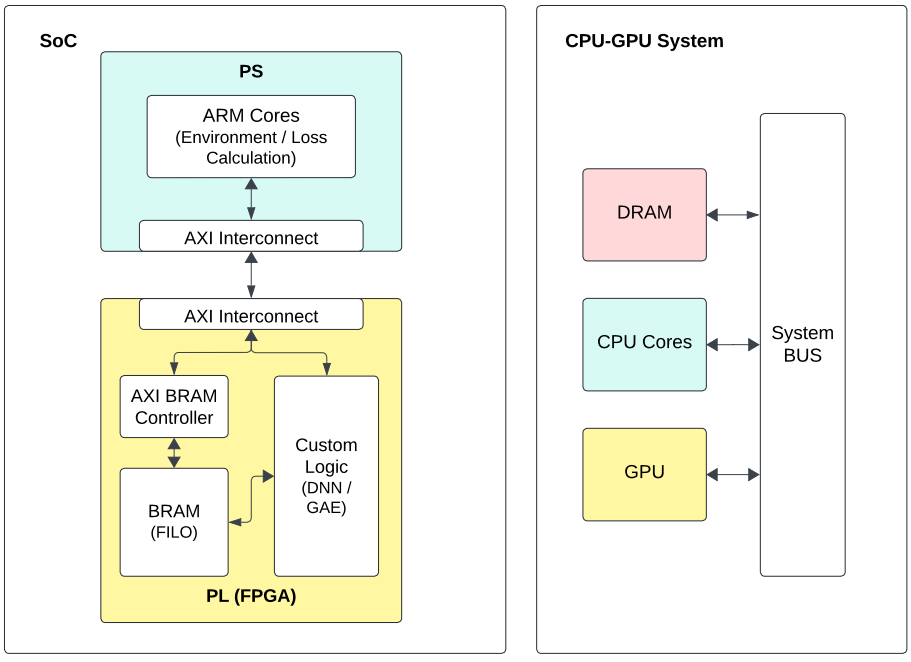

Figure 4.7: Integrated SoC Architecture for Full PPO Pipeline

4.3.2 Data Flow and Processing

The integration enables an efficient data flow that keeps critical data on-chip, reducing reliance on external DRAM and minimizing communication overhead. The PS runs the environment simulations locally, allowing the agent to interact with the environment using the current policy without incurring delays associated with external communication. States, actions, rewards, and value estimates are collected during simulation and stored in the on-chip Block RAMs (BRAMs).

The collected data is processed, standardized, and quantized according to the methods described in Section 4.4. This processed data is stored in the BRAMs, ready for further computation. The PL, containing the HEPPO accelerator, fetches the data from the BRAMs, performs de-quantization, and computes the advantages and rewards-to-go using the pipelined architecture and k-step lookahead approach. The computed advantages and rewards-to-go are written back to the BRAMs. Upon completion, the PL signals the PS.

For neural network inference and training, the systolic array implementation introduced by Meng et al. [32] or similar architectures can be adapted to leverage the FPGA's capabilities for high-throughput neural network computations. Their design achieves substantial performance improvements, ranging from 2.1× to 30.5× compared to state-of-the-art CPU implementations and 2× to 27.5× compared to CPU-GPU implementations. By integrating such a design, the PL handles both forward and backward passes required for policy and value function updates, reducing the computational burden on the PS and accelerating the training process.

This tightly integrated data flow confines data movements within the chip, leveraging the high-speed internal connections of the SoC. By minimizing off-chip communication, the architecture significantly reduces latency and enhances throughput, which is critical for real-time reinforcement learning applications.

4.3.3 Benefits and Potential Extensions

Integrating the full PPO pipeline on the SoC offers several significant benefits. On-chip communication reduces latency associated with data transfer between separate devices in traditional systems. By consolidating all components onto a single chip, the overhead of data transfer between CPU, GPU, and DRAM is eliminated, enhancing overall system performance. High-speed on-chip interconnects and memory architectures enable faster data access and processing, supporting the high throughput required for real-time applications. Additionally, SoC integration can lead to lower power consumption compared to systems relying on discrete CPUs and GPUs, making it suitable for embedded and portable applications.

The proposed solution opens the door for a fully hardware-accelerated PPO implementation, where the entire PPO pipeline, including environment simulation, GAE computation, and neural network training, is performed efficiently on-chip. The architecture is designed to be extensible, allowing for the incorporation of additional hardware accelerators and methodologies from related work. By incorporating advanced neural network accelerators, such as a systolic array design, the system's performance and adaptability to diverse reinforcement learning tasks can be further enhanced.

4.3.4 Challenges and Design Considerations

While integrating the full PPO pipeline on an SoC presents numerous advantages, it introduces challenges and design considerations. Resource constraints of FPGAs and SoCs, such as limited logic cells, DSP units, and BRAMs, require careful planning to balance resource allocation among environment simulation, HEPPO, neural network accelerators, and other components. Implementing complex environment simulations on hardware can be challenging, especially for environments with intricate physics or high-dimensional state spaces. Utilizing High-Level Synthesis (HLS) tools can facilitate the hardware implementation of environments by allowing designers to describe functionalities using high-level programming languages.

Designing custom hardware accelerators for neural network training adds complexity to the system. Ensuring correct and efficient operation of all components together requires meticulous hardware-software co-design and verification. Efficient memory management is critical, necessitating memory hierarchies and data compression techniques to optimize resource utilization. Coordinating timing between various hardware components and ensuring correct data synchronization is essential for system stability and performance. Implementing proper synchronization mechanisms between the PS and PL is crucial.

Moreover, the architecture must be adaptable to various reinforcement learning environments, which may have different computational requirements. Generalizing the solution to support a wide range of tasks involves careful abstraction and parameterization of hardware components.

4.4 Algorithm Modifications

To enable full integration of the PPO algorithm with HEPPO, several algorithmic modifications were necessary. These modifications aim to maintain the original training performance while adapting data storage and transfer methods to fit the hardware constraints and capabilities. This is primarily achieved by rescaling input data for HEPPO. Rescaling is crucial for two main reasons:

1. **Reducing Bandwidth Requirements**. A predictable and consistent data distribution facilitates effective quantization of rewards and value estimates. As illustrated experimentally in Figure 4.8, the value distributions vary over training iterations. Efficient quantization is necessary to optimize the use of on-chip memory resources (BRAMs) by reducing the data width multiple folds. This alleviates bandwidth constraints inherent in the hardware design and allows for more efficient data transfer.

2. **Generalizing the Solution**. Ensuring that the scale and distribution of rewards and value estimates are independent of specific environments and hyperparameters allows HEPPO to be a universal GAE solution. This uniformity is vital because reward distributions can vary by orders of magnitude across different environments, affecting the associated value estimates.

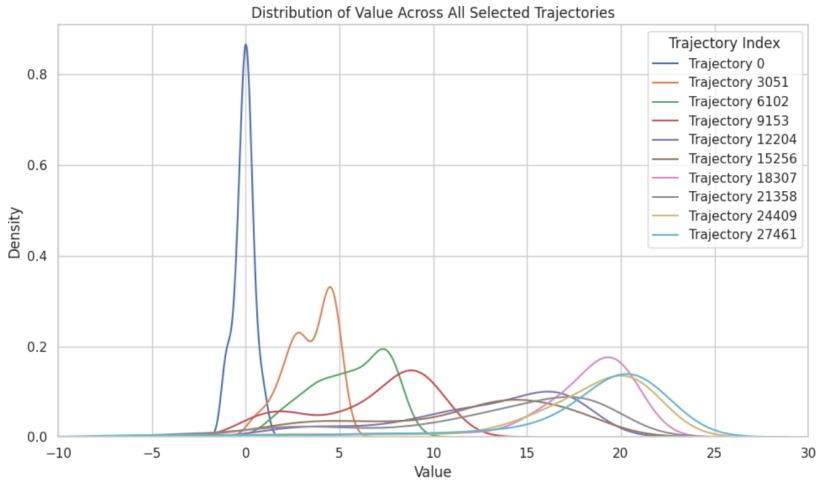

Figure 4.8: Distribution of value estimates across collected trajectories during training.

4.4.1 Data Standardization

Standardization techniques are employed to rescale the rewards and value estimates, making their distributions more uniform and bounded. Several methods were investigated to determine the most suitable approach for both rewards and value estimates.

Important Notation: In the context of standardization throughout the end of the thesis, the phrase "*One-way standardization*" refers to standardizing the data and later using it in the standardized form in computations, while the phrase "*Two-way standardization*" refers to standardizing the data and later de-standerdizing it back before using it in computations.

Normalization of Data

Normalization scales data to a specific range, typically between 0 and 1. In the context of PPO, each trajectory's rewards or value estimates can be normalized independently:

1. **Trajectory Collection**: Collect rewards or value estimates for a single trajectory.

2. **Compute Minimum and Maximum**: Determine the minimum (min) and maximum (max) values within the trajectory.

3. **Normalization**: Apply the normalization formula to each data point x:

$$x_{\text{norm}} = \frac{x - \min}{\max - \min}. \tag{4.4.1}$$

This method ensures that the data within each trajectory falls within the range [0, 1], facilitating uniform quantization and efficient storage. This method can be done as both One-way or Two-way.

Standardization of Data

Standardization involves scaling data to have a mean of zero and a standard deviation of one. For each trajectory, the following steps are performed:

1. **Trajectory Collection**: Collect rewards or value estimates for a single trajectory.

2. **Compute Mean and Standard Deviation**: Calculate the mean (μ) and standard deviation (σ) of the data within the trajectory.

3. **Standardization**: Apply the standardization formula to each data point x:

$$x_{\text{std}} = \frac{x - \mu}{\sigma}. \tag{4.4.2}$$

4. **Clipping**: To handle outliers and ensure data falls within a manageable range, apply clipping (e.g., to the range [-3, 3]) after standardization.

Standardization centers the data and scales it based on its variance, which can improve the efficiency of quantization and maintain the relative differences between data points. This method can be done in both one-way and two-way standardization.

Batch Standardization

Extending the concept of standardization of a trajectory vector, batch standardization was proposed for batches of vectors. Instead of processing each trajectory independently, a batch of consecutive trajectories is standardized together:

1. **Batch Collection**: Collect rewards or value estimates from multiple trajectories forming a batch.

2. **Compute Batch Mean and Standard Deviation**: Calculate the mean (μ_{batch}) and standard deviation (σ_{batch}) over the entire batch.

3. **Standardization**: Apply the standardization formula to each data point x in the batch:

$$x_{\text{std}} = \frac{x - \mu_{\text{batch}}}{\sigma_{\text{batch}}}. \tag{4.4.3}$$

4. **Clipping**: Apply clipping to ensure standardized data remains within a specified range.

Batch standardization leverages the statistical similarities across trajectories collected during the same training period, promoting consistency in data scaling and facilitating efficient quantization.

Block Standardization

Block Standardization (BS) extends the concept of batch standardization by aggregating data across multiple batches of trajectories from different agents into blocks. This concept is illustrated in Figure 4.9 where J_m^n is trajectory m collected by agent n. This approach leverages the collective information from all agents using the same policy, ensuring that the standardized data captures local statistical properties relevant to the current training context.

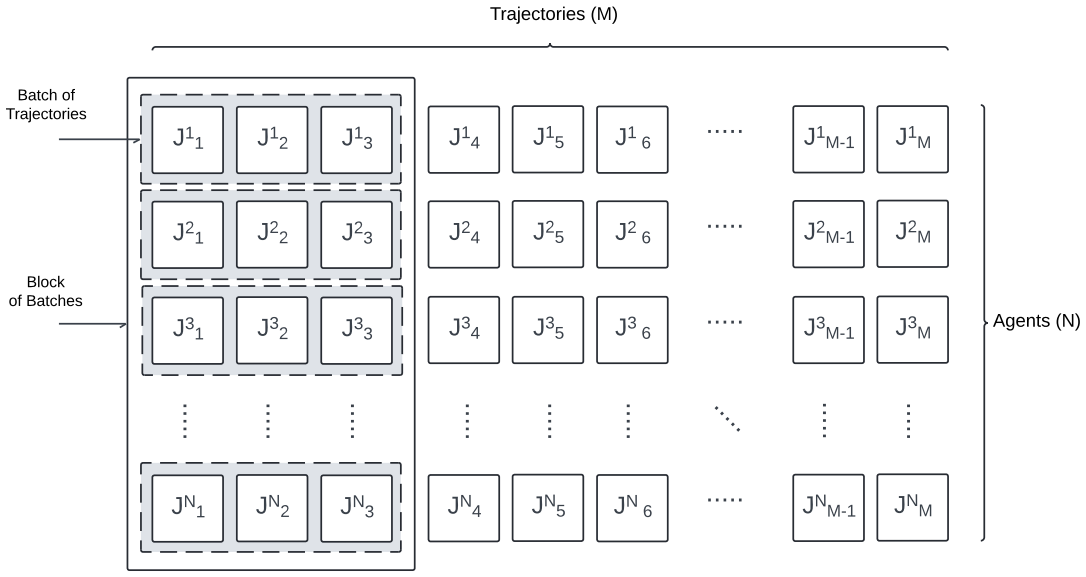

Figure 4.9: Batch and Block Standardization.

1. **Block Collection**: Aggregate rewards or value estimates from multiple batches of trajectories across all agents to form a block. This involves collecting data from a defined set of consecutive training iterations or time frames.

2. **Compute Block Mean and Standard Deviation**: Calculate the mean (μ_{block}) and standard deviation (σ_{block}) over the entire block. These statistics

are computed by considering all data points within the block, ensuring that the normalization reflects the local distribution of the data.

3. **Standardization**: Apply the standardization formula to each data point x in the block:

$$x_{\text{std}} = \frac{x - \mu_{\text{block}}}{\sigma_{\text{block}}}. \tag{4.4.4}$$

This step transforms the data to have a zero mean and unit variance within the block, facilitating consistent scaling across different contexts.

4. **Clipping**: Apply clipping to the standardized data to ensure that it remains within a specified range. Clipping helps in mitigating the impact of outliers and maintaining numerical stability during subsequent processing steps such as quantization.

BS effectively captures the local statistical properties of the data by grouping trajectories from similar contexts. This localized approach enhances the robustness of the standardization process, especially in environments where data distributions may vary over time or across different training phases. By preserving important data characteristics within each block, BS aids in efficient quantization and overall model performance.

Careful consideration is required to ensure that one-way standardization does not adversely affect the learning process, particularly when standardizing value estimates used as baselines in advantage calculations. Although one might propose always using two-way standardization—reverting data back to its original distribution after retrieval from memory—to avoid disrupting training, this approach would undermine a key objective: ensuring that the data's scale and distribution remain independent of

specific environments and hyperparameters to facilitate compatibility with the custom developed hardware. Therefore, one-way standardization is essential.

Dynamic Standardization

Dynamic Standardization (DS) is a novel One-way standardization technique that has been developed as part of the investigation of different standardization methods. The idea is that at each training epoch, reward standardization shall be conducted while accounting for all previously attained rewards. As it will be computationally and memory intensive to store and reprocess all the rewards across training, a more efficient way is to store a running mean and running standard deviation that gets updated every epoch with the new reward. To update the running mean with every new reward, the following equation is used:

$$\text{RunningMean}_n = \text{RunningMean}_{n-1} + \frac{r_n - \text{RunningMean}_{n-1}}{n}, \tag{4.4.5}$$

Where:

- n is the total number of rewards processed so far.
- r_n is the n-th reward.
- RunningMean_n is the running mean calculated up to the n-th reward.

As for the running standard deviation, inspired by Welford's algorithm [23, 46] for dynamically calculating variance over multiple iterations, the running variance for each new data point has been computed as follows.

1. Initialize M_0 and S_0 to 0.

2. For each new reward r_n

$$M_n = M_{n-1} + \frac{(r_n - M_{n-1})}{n}, \qquad \text{and} \tag{4.4.6}$$

$$S_n = S_{n-1} + (r_n - M_{n-1}) \times (r_n - M_n). \tag{4.4.7}$$

3. The running standard deviation after n rewards is then

$$\text{RunningSTD}_n = \sqrt{\frac{S_n}{n}}, \tag{4.4.8}$$

where M_n is the running mean after n rewards and S_n is the cumulative value used for calculating variance.

4.4.2 Data Quantization

Following standardization, data quantization is applied to reduce the precision of continuous rewards and value estimates by mapping them to a finite set of discrete values, referred to as quantization levels. Each quantization level represents a specific discrete value that corresponds to a range of input values, known as a quantization interval. This process simplifies the numerical representation of data, reducing memory usage and accelerating computation.

The effectiveness of quantization is typically evaluated by quantization loss, which measures the discrepancy between the original continuous values and their quantized representations.

Uniform Quantization

Uniform quantization divides the data range into equal intervals and maps each data point to the nearest quantization level. This method is straightforward to implement in hardware and works well in minimizing quantization error for uniformly distributed data, as the equal intervals provide even representation across the range.

1. **Define Quantization Range**: Determine the minimum (min) and maximum (max) values of the standardized data, which should be [-3, 3] after applying clipping to the standardized data.

2. **Select Bit Width**: Choose the number of bits b for quantization, determining the number of quantization levels $L = 2^b$.

3. **Compute Quantization Step Size**: Calculate the step size Δ:

$$\Delta = \frac{\max - \min}{L - 1}. \tag{4.4.9}$$

4. **Quantization**: Map each data point x to a quantized value q:

$$q = \text{round}\left(\frac{x - \min}{\Delta}\right). \tag{4.4.10}$$

   Here, q is an integer within the range $[0, L-1]$, representing one of the L quantization levels.

5. **Dequantization**: Reconstruct the approximate original value:

$$\hat{x} = q \times \Delta + \min. \tag{4.4.11}$$

Dequantization provides an approximation $\hat{x}$ of the original data point x, acknowledging that some information loss occurs during quantization.

Uniform quantization is effective when the data is uniformly distributed within the quantization range. However, its performance may degrade for non-uniform data distributions, where certain regions of the data range are more densely populated than others. In such cases, non-uniform quantization techniques can offer improved precision by allocating more quantization levels to regions with higher data density.

Non-Uniform Quantization

Non-uniform quantization enhances precision in regions where data points are densely populated by allocating more quantization levels near the mean of the data distribution and fewer levels towards the extremes. This method is particularly beneficial for data following a non-uniform distribution, such as the standard normal distribution. One way to achieve this is by applying the Gaussian Cumulative Distribution Function (CDF) to transform the data into a uniform distribution, performing uniform quantization on the transformed data, and then applying the inverse transformation to reconstruct the original values. Figure 4.10 illustrates how the CDF and its inverse (PPF) facilitate this transformation and reconstruction.

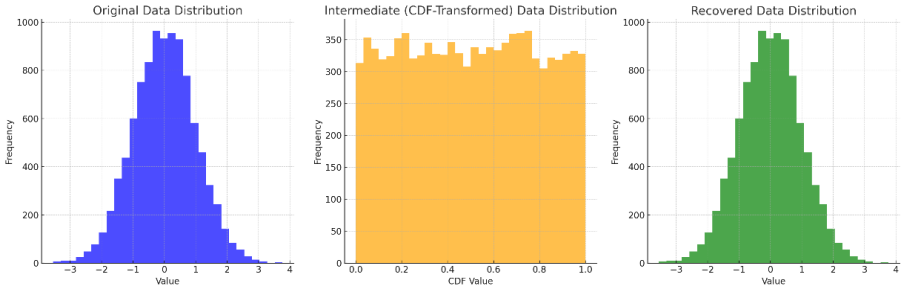

Figure 4.10: Empirical analysis on using CDF and PPF as Nonlinear Transformation in Non-Uniform Quantization.

1. **Standardize Data**: Ensure that the data follows a standard normal distribution ($\mu = 0, \sigma = 1$). Standardization is a crucial prerequisite for the effectiveness of the Gaussian CDF transformation.

2. **Apply Gaussian CDF Transformation**: Transform the standardized data using the Gaussian Cumulative Distribution Function (CDF) to map it to a uniform distribution over the interval $[0, 1]$:

$$r' = \Phi(r) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{r}{\sqrt{2}}\right)\right],$$

   where $\Phi(r)$ is the standard normal CDF and erf is the error function. This transformation ensures that the data is uniformly distributed, facilitating effective uniform quantization in the transformed space.

3. **Uniform Quantization in the Transformed Space**: Apply uniform quantization to the transformed values r' using an n-bit codeword.

   (a) **Select Bit Width**: Choose the number of bits b for quantization, determining the number of quantization levels $L = 2^b$.

(b) **Compute Quantization Step Size**: Calculate the step size Δ:

$$\Delta = \frac{1}{L-1}. \tag{4.4.12}$$

(c) **Quantization**: Map each transformed data point r' to a quantized value q:

$$q = \text{round}\left(r' \times (L-1)\right). \tag{4.4.13}$$

Here, q is an integer within the range $[0, L-1]$, representing one of the L quantization levels.

4. **Store Quantized Values**: The quantized values q are stored efficiently using b-bit codewords, significantly reducing memory usage while maintaining essential data characteristics.

5. **Decoding via Inverse Transformation**: To reconstruct the original data, apply the inverse transformation using the Quantile Function (PPF), which is the inverse of the Gaussian CDF:

$$\Phi^{-1}(p) = \sqrt{2}\,\text{erf}^{-1}\left(2p-1\right),$$

where $p = \frac{q}{L-1}$ is the normalized quantized value. The dequantized value $\hat{r}$ is then obtained as:

$$\hat{r} = \Phi^{-1}\left(\frac{q}{L-1}\right). \tag{4.4.14}$$

This step accurately reconstructs the original standardized data r from the

quantized values q, leveraging the properties of the Gaussian CDF and its inverse.

Non-uniform quantization offers enhanced precision by allocating more quantization levels near the mean of the data distribution, thereby achieving higher accuracy in densely populated regions while maintaining efficient storage through reduced bit usage. This method is particularly effective for data that deviates from a uniform distribution, as it better preserves important data characteristics. However, it relies on the assumption that the data follows a standard normal distribution and introduces additional computational steps due to the application of the Gaussian CDF and its inverse. Despite these considerations, non-uniform quantization provides a balanced approach to minimizing quantization error in regions of higher data density while conserving memory resources.

Block-Based Quantization Methods

Block-based quantization is a technique that compresses a block of values by sharing common quantization parameters among the elements within the block. This method significantly reduces memory usage while maintaining minimal round-off errors. There are two primary types of block-based quantization:

1. **Shared Scaling Factor**: All values in a block are scaled using a common mean and standard deviation. This approach aligns with block standardization, enabling uniform quantization within the block.

2. **Shared Exponent Bias**: Values within a block share a common exponent or exponent bias, allowing for efficient integer representations and simplifying arithmetic operations.

Several block-based quantization methods were explored:

- **Block Floating-Point (BFP)**:
  - Shares a common exponent across all values in a block.
  - Suitable for blocks with low variance, effectively bounding the representational range [9].
- **Block MiniFloat (BM)**:
  - Utilizes a shared exponent bias within a block.
  - Balances high precision and range, though it may introduce larger quantization errors for medium values [15, 19].
  - Efficient for representing values concentrated near peaks in multimodal distributions.
- **Block Logarithm (BL)**:
  - Represents values as powers of two by sharing a common exponent bias and setting the mantissa to a fixed value.
  - Ideal for data with large dynamic ranges, simplifying multiplication and division to bitwise shifts and additions.
  - Provides a non-linear quantization scale, effectively handling large values and dynamic ranges.
- **MiniFloat and Denormalized MiniFloat (DMF)**:
  - MiniFloat offers a compact floating-point representation with fewer bits for the exponent and mantissa.

– DMF enhances MiniFloat by using zero as the implicit leading bit in the mantissa, improving precision near zero.

– Both methods sacrifice some precision and range for reduced memory usage, making them suitable for memory-constrained applications.

Figure 4.11 provides an illustration of the various block-based quantization methods discussed.

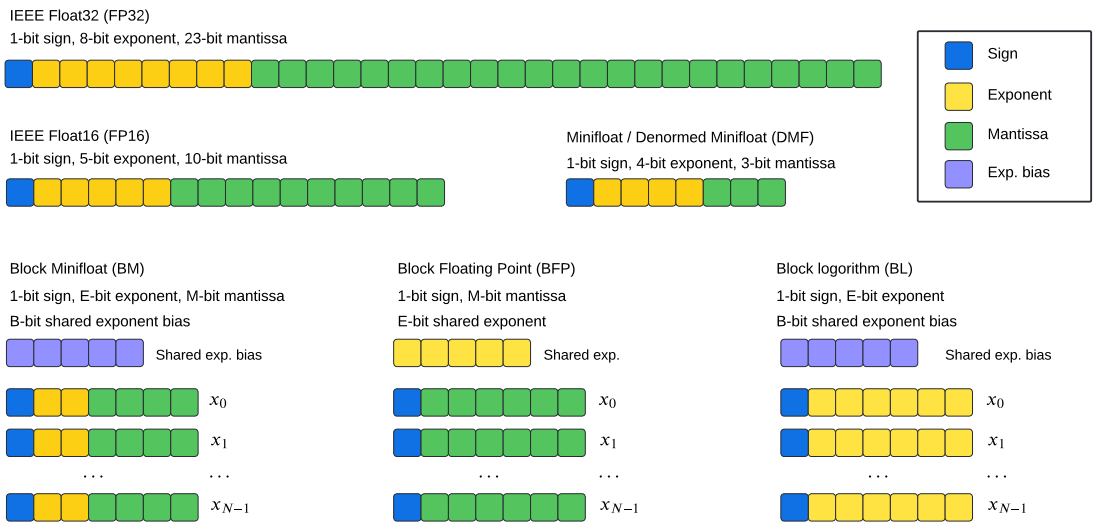

Figure 4.11: Illustration of Block-Based Quantization Methods. Original illustration based on Zhang at al [51].

Advantages and Considerations Block-based quantization methods offer significant memory and computational savings by sharing quantization parameters across data blocks, thereby reducing the number of bits required per value and simplifying arithmetic operations. Additionally, techniques like Block Floating-Point (BFP) adapt to local data characteristics, maintaining precision where it is most needed.

However, these methods introduce challenges such as increased complexity in managing shared parameters, the potential for larger quantization errors in high-variance blocks, and the necessity for more sophisticated hardware implementations.

4.4.3 Summary

This chapter provides a structured approach to enhancing PPO's performance for constrained hardware environments. Beginning with a detailed time profiling analysis, the study identifies GAE computation and environment simulation as primary bottlenecks. With this insight, the HEPPO accelerator is designed to optimize GAE computation through a pipelined architecture and k-step lookahead techniques, enabling concurrent trajectory processing. Additional adjustments, including data standardization, quantization, and organizing data flow to match the hardware's FILO memory access pattern, make the PPO algorithm more compatible with the proposed hardware accelerator. These combined optimizations aim to improve PPO's scalability and real-time performance, facilitating deployment in hardware-limited applications.

Chapter 5

Results and Discussion

This section presents the results of the algorithmic modifications and hardware implementation strategies aimed at optimizing the Proximal Policy Optimization (PPO) algorithm. The analysis focuses on two primary dimensions: the impact of algorithmic modifications on training performance and the benefits achieved through hardware implementation using the Hardware-Efficient Proximal Policy Optimization (HEPPO) accelerator.

5.1 Algorithmic Modifications

The algorithmic modifications centered on data standardization and quantization techniques designed to stabilize training and facilitate efficient hardware implementation. Specifically, different standardization methods for rewards and value estimates, as well as various quantization methods, were investigated to assess their effects on PPO's performance.

5.1.1 Impact of Reward Standardization Techniques

Experiments were conducted to evaluate the effectiveness of different reward standardization methods, comparing five PPO setups in the *Humanoid* environment [13]:

1. **Base PPO**: The standard PPO algorithm without reward standardization.
2. **One-way Block Standardization (BS)**: Rewards are standardized using BS and used in their standardized form in computations.
3. **Two-way Block Standardization (BS)**: Rewards are standardized using BS and de-standardized back before computations.
4. **One-way Dynamic Standardization (DS)**: Rewards are standardized using DS and used in their standardized form.
5. **One-way Dynamic Standardization (DS) with Advantage Standardization**: Rewards are standardized using DS, and the computed advantages are also standardized.

Details of the neural network architectures and hyperparameters used can be found in Section 4.1.1.

Figure 5.1 illustrates the average rewards over training epochs for these setups, using a rolling average of 800 readings to smooth the data.

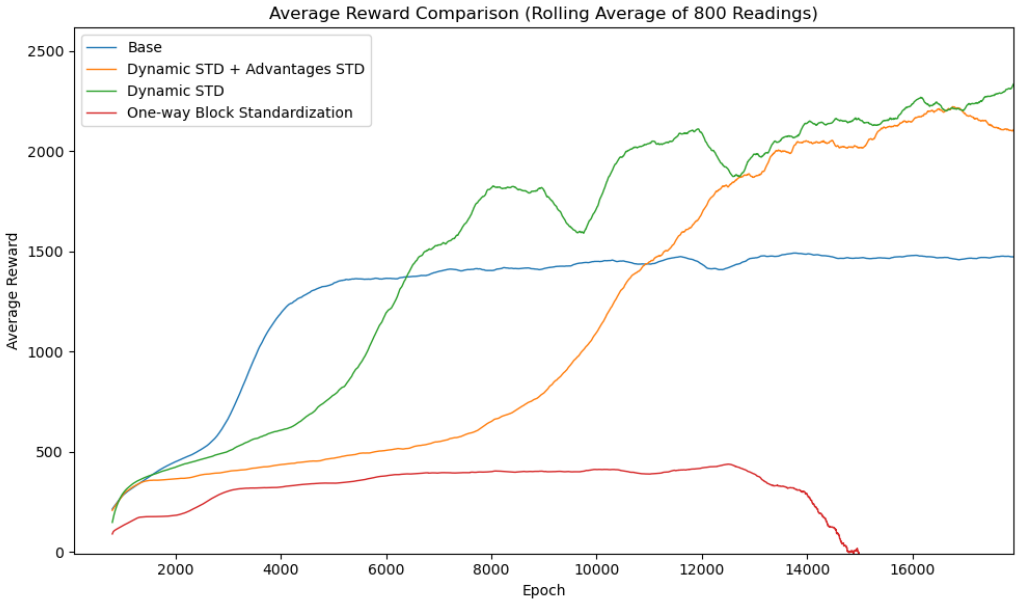

Figure 5.1: Comparison of average rewards for different reward standardization techniques in the Humanoid environment.

In the **Base PPO** setup (blue line), the algorithm demonstrates rapid initial improvement, with the average cumulative reward stabilizing around 1,400 after approximately 6,000 epochs. This plateau indicates that the agent has reached its learning capacity under the given conditions.

The **One-way BS** setup (red line) shows a slow initial performance increase, reaching a low plateau at around 3,000 epochs. Subsequently, the average reward gradually declines, with significant decreases observed after 13,000 epochs. This decline suggests that One-way BS leads to training divergence. By altering the reward distribution within each trajectory independently, block by block, One-way standardization disrupts the relative differences in rewards across epochs. This disruption equalizes rewards from good and bad policies, misleading the training process and preventing the agent from accurately discerning effective actions. Consequently, the

learning algorithm struggles to make meaningful updates, leading to instability and performance degradation over time.

Conversely, the **Two-way BS** setup (not shown in the figure for simplicity) mirrors the behavior of the Base PPO. In Two-way standardization, rewards are de-standardized back to their original distribution before use in computations, preserving the inherent reward structure. While this method facilitates quantization and minimizes memory usage, it fails to achieve one of the primary objectives: ensuring that the scale and distribution of rewards are independent of specific environments and hyperparameters. In certain environments or with particular hyperparameter configurations, the reward scale could be excessively large, potentially causing hardware overflow issues. Additionally, the effectiveness of different quantization techniques, discussed later, may not be consistent if reward distributions vary significantly.

These challenges motivated the development of the **DS** technique, as detailed in section 4.4.1. The **One-way DS** setups (green and orange lines in Figure 5.1) exhibit a strong improvement in average rewards, surpassing the Base PPO around 6,000 and 11,000 epochs, respectively. The average rewards continue to increase steadily, reaching approximately 2,500 before plateauing—about **50%** higher than the Base PPO plateau. This significant enhancement is achieved with minimal computational overhead, involving only the maintenance of running statistics for the rewards. The results indicate that DS effectively stabilizes training and enhances learning, enabling the agent to achieve substantially higher performance levels compared to both the Base PPO and the BS approach.

The orange line represents the **One-way DS with Advantage Standardization** setup. Standardizing the computed advantage vectors is a common practice to

stabilize gradient updates and ensure smoother training, as highlighted in various implementations [31, 48]. However, while this addition resulted in smoother training, it also led to a slower increase in cumulative rewards.

Based on these findings, DS (with no Advantage Standardization) emerges as the preferred method for standardizing rewards in PPO. It effectively stabilizes training, enhances learning efficiency, and leads to significantly higher cumulative rewards with minimal additional overhead. To validate the generality of this conclusion, additional experiments were conducted in different environments.

5.1.2 Evaluation in the Lunar Lander Environment

Figure 5.2 presents the results of applying DS in the *Lunar Lander* environment from Gymnasium [14], which is described in Section 4.1.1. In this environment, the DS-enhanced approach demonstrates faster initial learning, with the average reward surpassing zero nearly 50 epochs earlier than the Base PPO. The performance consistently exceeds that of the Base PPO algorithm. While the Base PPO plateaus at an average reward of around 6,000, the DS approach reaches a stable average reward of approximately 10,000, representing a significant 67% increase in cumulative rewards. This improvement is achieved with minimal computational overhead, reinforcing the effectiveness of DS across different environments.

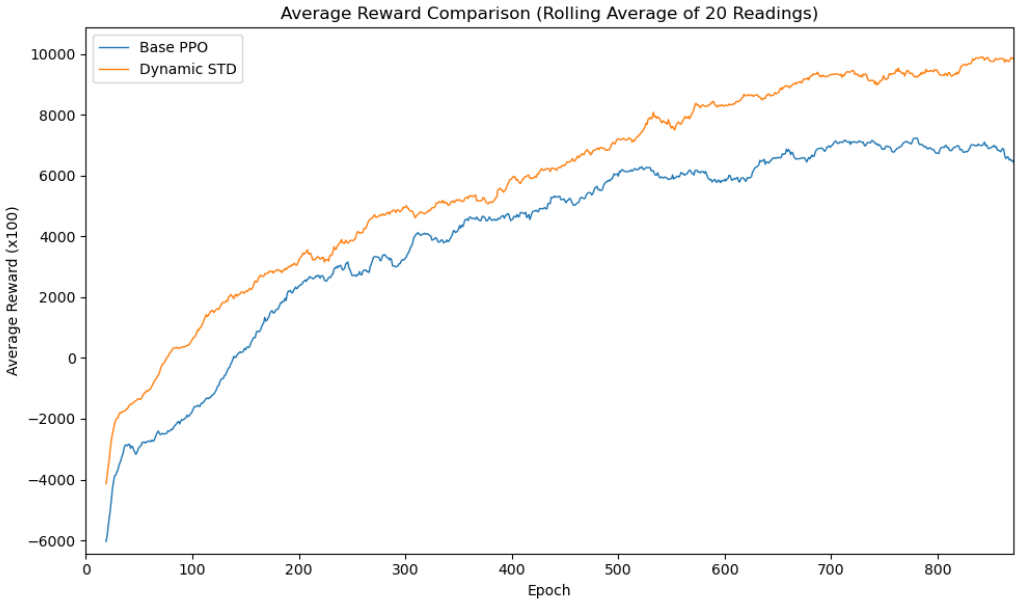

Figure 5.2: Effect of DS on average rewards in the Lunar Lander environment.

5.1.3 Standardization of Value Estimates

The investigation into standardization techniques also considered their application to value estimates generated by the critic network. It was observed that all forms of standardization applied to value estimates led to training divergence. value estimates serve as baselines in advantage calculations and are sensitive to shifts in distribution. Interestingly, analysis and experimental results indicate that explicit standardization of value estimates is unnecessary because they are implicitly standardized through the DS of rewards.

The value estimates produced by the critic neural network are predictions of the expected return at a given state, trained on dynamically standardized rewards. As shown in Figure 5.3, collected data reveals that the distribution of value estimates across different trajectories in various environments using dynamic standardization

of rewards consistently centers around zero over time. This behavior suggests that the network indirectly learns to standardize the value estimates, achieving similar outcomes without explicit intervention.

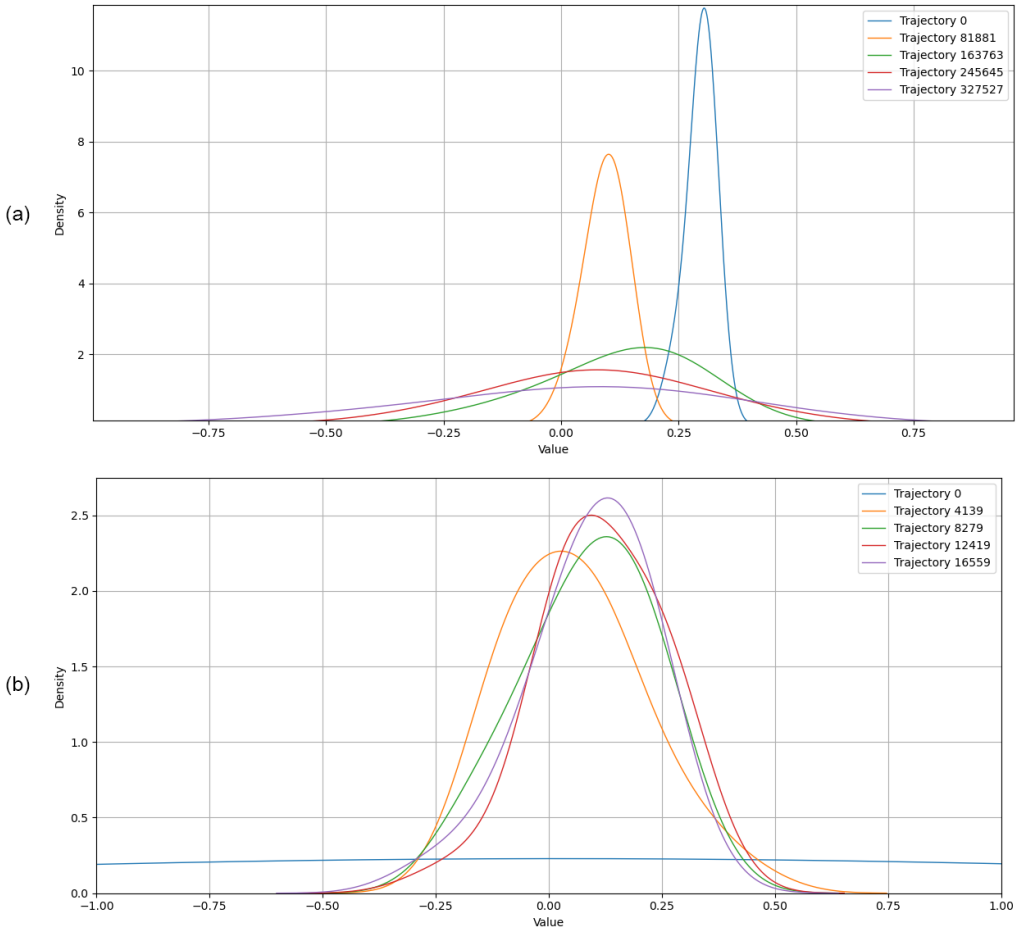

Figure 5.3: Distribution of value estimates for selected trajectories across training in (a) Humanoid environment and (b) Lunar Lander environment with rewards x100.

In both environments, the value distributions consistently center around zero, with varying levels of spread depending on the environment and trajectory. In the Humanoid environment, the distributions exhibit broader peaks and greater variability, while in the Lunar Lander environment, the distributions are more symmetrical

and concentrated. This pattern supports the conclusion that DS effectively stabilizes value estimates without causing significant shifts over time.

5.1.4 Impact of Quantization Techniques

To optimize the PPO algorithm for hardware efficiency, several quantization methods for rewards and value estimates were explored:

1. **Uniform Quantization**
2. **Non-Uniform Quantization (CDF-based)**
3. **Block Floating-Point Quantization**
4. **Block Logarithmic Quantization**
5. **Base PPO with DS (No Quantization)**

The chosen bit-width for quantization in the presented experiments was 8 bits, allowing for a fair comparison of the different quantization methods in terms of their impact on training performance and computational efficiency. Figure 5.4 compares the average rewards over training epochs for these methods.

Details of the neural network architectures and hyperparameters used can be found in Section 4.1.1.

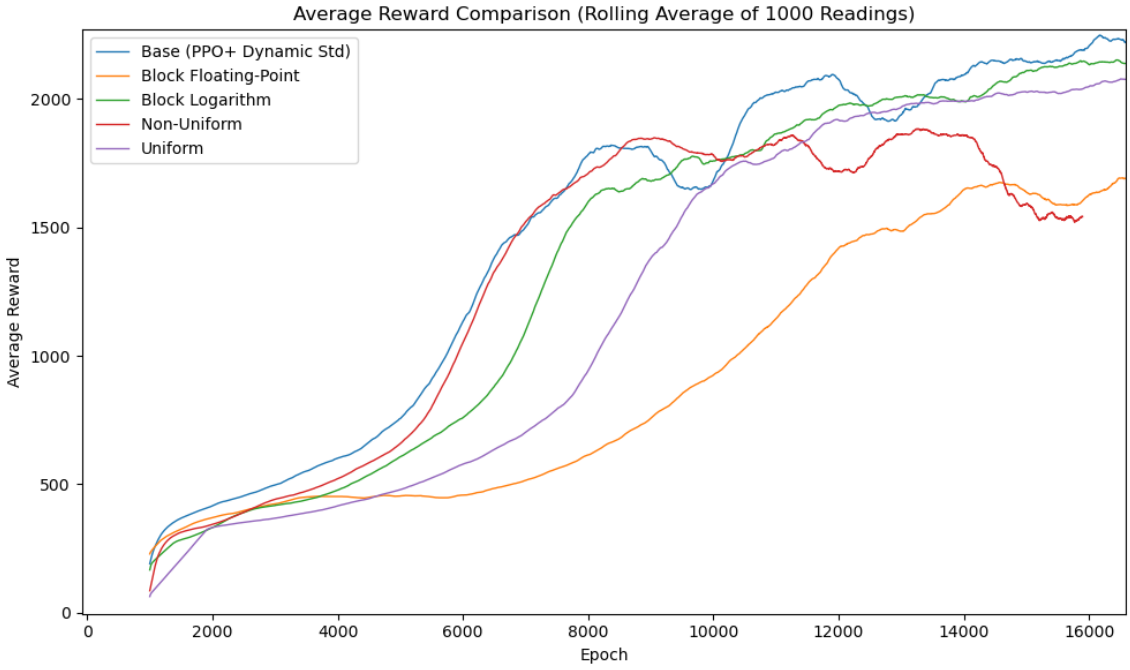

Figure 5.4: Comparison of average rewards for different quantization techniques.

In the **Base PPO with Dynamic Standardization (DS)** setup (blue line), serving as the control, DS is applied without quantization. The average reward improves gradually throughout training, achieving approximately 2,300 in later epochs, indicating superior long-term performance.

The **Uniform Quantization** method (purple line) employs equal-sized intervals to quantize standardized rewards and value estimates. It provides steady performance, reaching an average reward of approximately 2,000 towards the end of training, with a positive slope suggesting potential for further improvement. While slightly trailing the performance of Base PPO with DS, this method stands out for its simplicity and ease of implementation.

The **Non-Uniform Quantization** method (red line) utilizes a Gaussian CDF to allocate more quantization levels near the data mean. While it initially achieves

a faster increase in average rewards compared to uniform quantization, it quickly reaches a plateau at an average reward below 2,000 mid-training. Its performance further deteriorates in later epochs. Despite its theoretical advantages, the added computational complexity and observed performance collapse over extended training make it an unsuitable choice.

The **Block Logarithmic Quantization** method (green line) uses a logarithmic scale for quantization, effectively handling large dynamic ranges. It quickly reaches high rewards, peaking around 2,200, and outperforms other quantization methods early in training. However, its performance nearly converges with uniform quantization in later stages, reducing its long-term advantage.

The **Block Floating-Point Quantization** method (orange line) applies a shared exponent across blocks, reducing memory usage. It underperforms, plateauing slowly around 1,500 average rewards, indicating significant precision loss and suboptimal learning.

An additional observation from the experiments in Figure 5.4 is the smoother learning curves exhibited by the quantized methods compared to the baseline PPO with Dynamic Standardization (DS). This suggests that quantization reduces noise in the value estimates and rewards, thereby stabilizing the training process. Such stability is particularly advantageous in hardware implementations and underscores the broader benefits of quantization beyond mere computational efficiency.

5.1.5 Discussion of Quantization Results

The experimental results highlight several key observations:

- **Uniform Quantization Balances Performance and Efficiency**: Achieving

average rewards similar to more complex methods, uniform quantization offers simplicity, computational efficiency, and ease of hardware implementation. It involves simple operations with constant time complexity per element, resulting in low computational overhead. Basic arithmetic operations are straightforward to implement using standard hardware components.

- **Limited Benefits from Non-Uniform Quantization**: Although non-uniform quantization theoretically offers advantages by allocating more quantization levels to densely populated regions, it fails to provide significant performance gains over uniform quantization. Its added computational complexity, combined with performance deterioration over extended training, makes the increased implementation challenges unjustifiable.

- **Diminishing Returns of Block Logarithmic Quantization**: While it excels early in training, block logarithmic quantization's performance converges with uniform quantization over time. This reduces its long-term advantage, and the increased complexity may not be warranted for the initial performance boost.

- **Precision Loss in Block Floating-Point Quantization**: The significant precision loss leads to poor performance, making block floating-point quantization unsuitable for tasks requiring high numerical accuracy.

- **Trade-Off with No Quantization**: The Base PPO with DS achieves the highest rewards but demands greater memory capacity and bandwidth to keep the hardware unit fully utilized, highlighting a trade-off between resource efficiency and performance. However, the performance gains are minimal compared

to the significant resource savings achieved with quantization.

Based on the findings, uniform quantization is preferred due to its computational efficiency, robustness, memory efficiency, and comparable performance. It employs simple arithmetic operations, resulting in low computational overhead and ease of hardware implementation. Additionally, uniform quantization is less sensitive to data variations, ensuring robustness across different environments and training scenarios.

5.1.6 Optimal Bit Width for Uniform Quantization

Selecting the optimal bit width for uniform quantization is critical to balance memory efficiency and training performance in the PPO algorithm. Extensive experiments were conducted with bit widths ranging from 3 to 10 bits to assess their impact. Figures 5.5 and 5.6 illustrate the average rewards over training epochs for different bit widths.

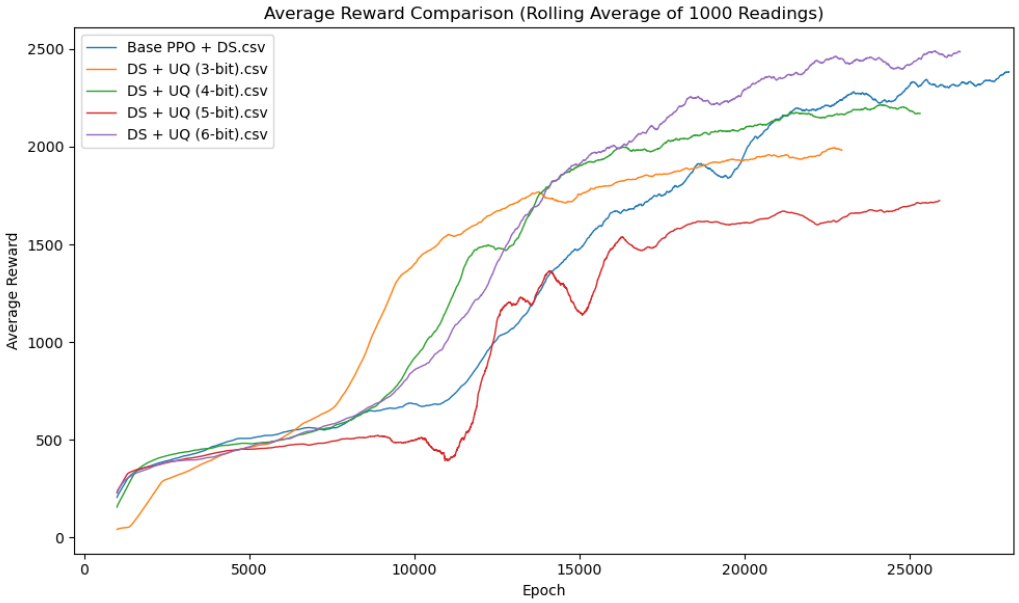

Figure 5.5: Uniform quantization of rewards using 3 to 6 bits.

Quantization with lower bit widths, specifically 3 to 7 bits, exhibited significant instability and inconsistent performance. Approximately 10 to 15 trials were conducted for each bit width to account for the stochastic nature of reinforcement learning. Agents quantized with 3 or 4 bits demonstrated erratic behavior—sometimes matching the baseline performance but often failing to learn effectively. Bit widths of 5 and 7 bits did not consistently outperform the lower bit widths, with performance fluctuating widely across trials. This instability stems from the coarse discretization at low bit widths, which introduces substantial quantization noise, distorts reward signals, and disrupts the learning process inherent in policy gradient methods.

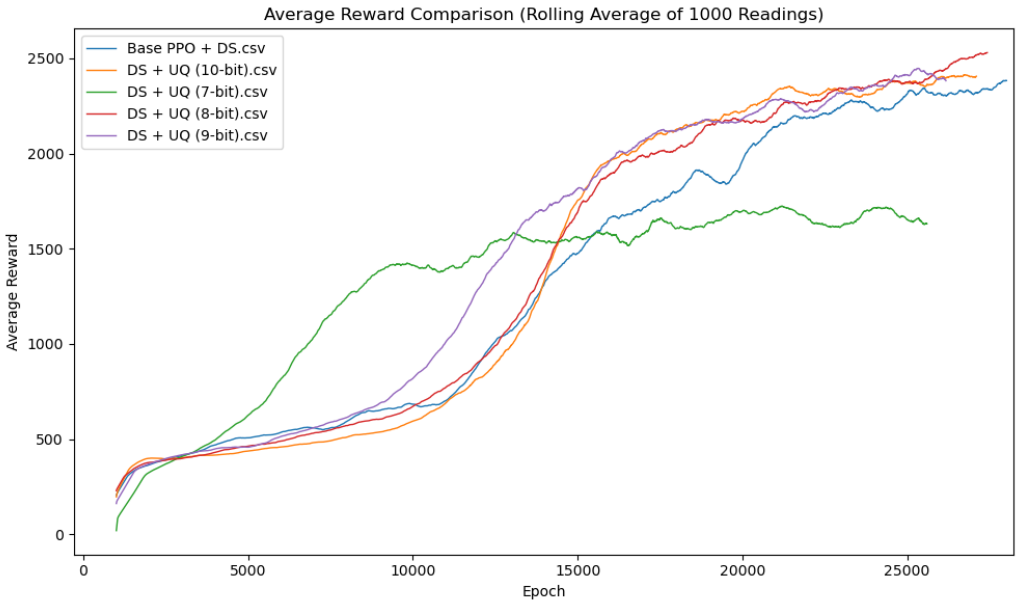

Figure 5.6: Uniform quantization of rewards using 7 to 10 bits.

In contrast, quantizing with 8 bits or more consistently achieved stable learning and high performance, closely matching the Base PPO with DS. Agents with 8 to 10 bits demonstrated reliable convergence and maintained high average rewards across all trials. The 8-bit precision effectively preserves essential information in the rewards

and value estimates, enabling the agent to learn efficiently while significantly reducing memory usage.

These findings indicate that 8 bits is a threshold for stable uniform quantization that ensures both training stability and hardware efficiency. The increased precision at 8 bits mitigates the quantization errors that adversely affect learning at lower bit widths. Therefore, selecting a bit width of 8 bits or higher is recommended to avoid the instability observed with 3 to 7 bits and to maintain high performance in PPO training.

5.2 Hardware Implementation Results

To evaluate the practical benefits of the proposed algorithmic modifications and demonstrate the feasibility of deploying the PPO algorithm on specialized hardware, the HEPPO accelerator was implemented on the AMD-Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. This section presents the hardware implementation details, resource utilization, memory requirements, and performance evaluation of the HEPPO accelerator.

5.2.1 Implementation Details

A parameterized Verilog model of the HEPPO pipelined architecture, as described in subsection 4.2.1, was developed with a data width of 32 bits after de-quantization and implemented on the ZCU106 Evaluation Kit.

The microarchitecture consists of multiple Processing Elements (PEs) configured in a pipelined fashion to process data in parallel. Each PE implements the k-step

lookahead approach for efficient computation of advantages and rewards-to-go. The design emphasizes continuous data flow and efficient memory usage to achieve high throughput and low latency.

5.2.2 Memory Utilization and Bandwidth Requirements

Efficient memory utilization and bandwidth management are critical for high-performance hardware accelerators. In the HEPPO accelerator, on-chip BRAMs are used to store rewards, value estimates, advantages, and rewards-to-go. To support a typical large-scale reinforcement learning setup with 64 agents (trajectories), each consisting of 1,024 timesteps, and using pairs of 8-bit quantized rewards and value estimates, the following calculations were performed:

Memory Capacity The total memory required to store the full vectors of rewards and value estimates for all agents is calculated as:

$$\text{Memory Size} = 64 \text{ agents} \times 1,024 \text{ timesteps} \times 8 \text{ bits} \times 2 = 128 \text{ KB} \tag{5.2.1}$$

This calculation accounts for both rewards and value estimates, and since advantages and rewards-to-go overwrite the same memory locations during processing, no additional memory is needed for them. Each BRAM block on the ZCU106 provides 36 Kb (4.5 KB) of storage. Therefore, the number of BRAM blocks required is:

$$\text{Number of BRAMs} = \frac{128 \text{ KB}}{4.5 \text{ KB}} \approx 29 \text{ BRAMs} \tag{5.2.2}$$

This constitutes approximately 9% of the available 312 BRAM blocks on the ZCU106, ensuring that the memory capacity requirements are well within the device's capabilities.

Bandwidth Requirements To keep all the parallel Processing Elements (PEs) fully utilized and ensure continuous data flow, the bandwidth requirements must be met. The bandwidth required per clock cycle for reading rewards and value estimates is:

$$\text{Read Bandwidth} = 64 \text{ agents} \times 8 \text{ bits} \times 2 = 128 \text{ bytes per cycle} \tag{5.2.3}$$

An additional 128 bytes per cycle are required for writing back the computed advantages and rewards-to-go:

$$\text{Write Bandwidth} = 128 \text{ bytes per cycle} \tag{5.2.4}$$

Therefore, the total bandwidth requirement is:

$$\text{Total Bandwidth} = \text{Read Bandwidth} + \text{Write Bandwidth} = 256 \text{ bytes per cycle} \tag{5.2.5}$$

Each dual-port BRAM on the ZCU106 can handle 4 bytes per port per cycle. To

meet the total bandwidth requirement, the number of BRAM ports needed is:

$$\text{Number of BRAM Ports} = \frac{256 \text{ bytes per cycle}}{4 \text{ bytes per port per cycle}} = 64 \text{ ports} \tag{5.2.6}$$

Since each dual-port BRAM provides two ports, the number of BRAM blocks required is:

$$\text{Number of BRAM Blocks} = \frac{64 \text{ ports}}{2 \text{ ports per BRAM}} = 32 \text{ BRAMs} \tag{5.2.7}$$

This constitutes approximately 10% of the available BRAM resources. The calculations confirm that the ZCU106 Evaluation Kit has sufficient BRAM blocks to meet both the memory storage and bandwidth needs, allowing efficient parallel processing and avoiding memory bottlenecks.

5.2.3 Area Utilization

The resource utilization of the HEPPO accelerator was analyzed for different configurations of the lookahead steps (n) in the PEs. Figure 5.7 illustrates the resource utilization percentages for implementations with 1-step, 2-step, and 3-step lookahead per PE.

As shown in the figure, there is a quadratic increase in resource usage with each increment in n. The increase in lookahead steps impacts the utilization of various resources, including Look-Up Tables (LUTs), Flip-Flops (FFs), and Digital Signal Processing (DSP) units.

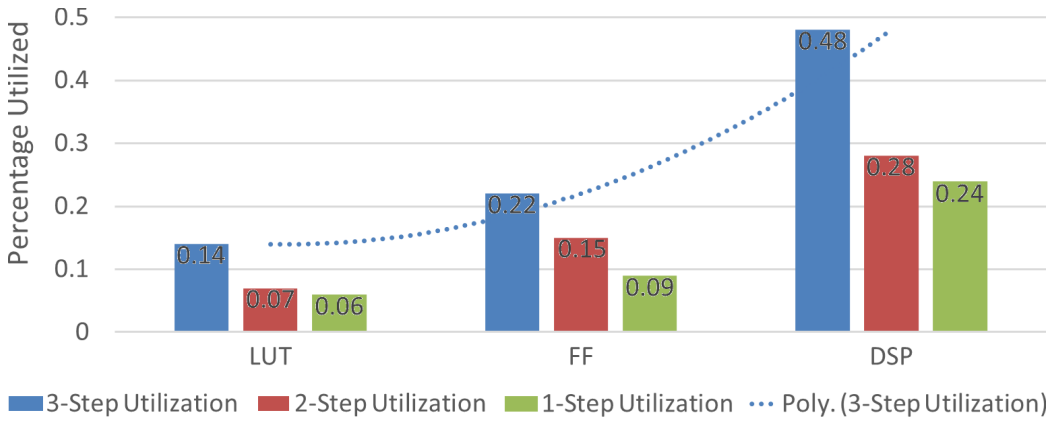

Figure 5.7: Resource utilization percentages for different lookahead steps (n) per Processing Element (PE).

Based on implementations, it was demonstrated that configuring $n > 1$ enabled the system to achieve its maximum operating frequency of 300 MHz, which represents the peak frequency attainable on the current device. This marks a significant improvement over the baseline Processing Element (PE), which operated at approximately 275 MHz, achieved through intensive pipelining and the elimination of pipeline stalls. Notably, the k-step lookahead methodology is designed to scale effectively, and on more advanced FPGA platforms with higher frequency capabilities, it has the potential to unlock even greater performance. A 2-step lookahead configuration was ultimately selected, providing an optimal balance between enhanced performance and resource utilization.

Table 5.1 presents the resource utilization for a system with 64 PEs configured with a 2-step lookahead. The utilization percentages indicate that the ZCU106 Evaluation Kit can comfortably accommodate the HEPPO accelerator without exceeding resource limitations.

Table 5.1: Resource utilization for a 2-step lookahead system with 64 Processing Elements (PEs).

Resource	**Total Usage**	**Available**	**Utilization (%)**
LUTs	12,864	274,080	4.69
FFs	54,336	548,160	9.91
DSPs	768	2,520	30.48

The most significant utilization is in the DSP units at approximately 30.48%. The efficient resource usage ensures that the system can run at the desired frequency and handle the required throughput without encountering resource constraints.

5.2.4 Performance Evaluation

The HEPPO accelerator's performance was evaluated in terms of processing speed and its impact on overall training time. The accelerator operates at a clock frequency of 300 MHz. With the 2-step lookahead configuration and due to the intensive pipelining and absence of pipeline stalls, a single PE can process one element per clock cycle. Therefore, the processing capability per PE is:

$$\begin{aligned} \text{Processing Rate per PE} &= 300 \text{ MHz} \times 1 \text{ element per cycle} \\ &= 300 \text{ million elements per second} \end{aligned} \tag{5.2.8}$$

For a system with 64 PEs, the total processing capability is:

$$\begin{aligned} \text{Total Processing Rate} &= 300 \text{ million elements per second per PE} \times 64 \text{ PEs} \\ &= 19.2 \text{ billion elements per second} \end{aligned} \tag{5.2.9}$$

To compare this performance with traditional CPU-GPU systems, tests were conducted using a standard GAE implementation [48] on a system comprising 32 Intel Xeon Silver 4216 CPU cores at 2.10 GHz and a Tesla V100-SXM2-32GB GPU. This setup achieved a processing rate of approximately 9,000 elements per second for the GAE computation phase. The substantial difference in processing rates indicates that the HEPPO accelerator can process elements over two million times faster than the traditional implementation.

This significant speedup is attributed to the custom hardware design optimized for the GAE computation, the elimination of memory access latency due to on-chip BRAM usage, and the ability to process multiple trajectories in parallel. Additionally, the HEPPO accelerator reduces communication overhead between the CPU and GPU, which is a common bottleneck in traditional systems.

By integrating the HEPPO accelerator into the PPO training pipeline, the time taken for the GAE computation phase, which traditionally accounts for up to 37% of the total training time, becomes a fraction of 1%. This results in an overall training time reduction and enhances the practicality of using PPO in real-time and resource-constrained applications.

5.3 Summary

The hardware implementation of the HEPPO accelerator demonstrates the feasibility and advantages of deploying the modified PPO algorithm on specialized hardware. The efficient resource utilization, substantial speedup in processing, and effective memory management highlight the potential for significant performance gains. The accelerator's design allows for scalability and adaptability, making it suitable for a wide range of reinforcement learning tasks and environments. By reducing the GAE computation time up to approximately 37% of the total training time to a negligible fraction, the overall training process becomes significantly more efficient. These results validate the proposed algorithmic modifications and support the integration of reinforcement learning algorithms into embedded systems and real-time applications.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

This thesis presents a comprehensive approach to optimizing the Proximal Policy Optimization (PPO) algorithm for hardware efficiency, addressing both algorithmic enhancements and hardware implementation strategies. The key contributions of this work are summarized below:

Algorithmic Enhancements

- **Introduction of Dynamic Standardization for Rewards**: Developed a novel dynamic standardization technique that stabilizes learning and enhances training performance across various environments. Dynamic standardization resulted in up to a **67% increase in cumulative rewards** in some environments compared to standard methods, demonstrating a significant impact on overall learning efficiency.

- **Extensive Analysis and Implementation of Standardization and Quantization Techniques**: Performed an in-depth evaluation of various standardization methods (e.g., batch standardization, block standardization, and dynamic standardization) and quantization strategies (uniform, non-uniform, and block quantization). The analysis culminated in the implementation of **8-bit uniform quantization** for rewards and value estimates, reducing memory requirements by **75%** without compromising training performance. These techniques are essential for hardware-efficient implementations, enabling the storage and transfer of a large number of trajectories within constrained on-chip memory resources.

Hardware Implementation Strategies

- **Design of HEPPO: A Highly Parallelized Architecture for GAE Computation**: Developed the HEPPO microarchitecture, which processes collected trajectories concurrently, enabling parallel computation of advantages and rewards-to-go. By employing 64 Processing Elements (PEs), HEPPO achieves a processing capability of **19.2 billion elements per second**, significantly accelerating the Generalized Advantage Estimation (GAE) computation phase.

- **Implementation of a k-Step Lookahead Approach for Intensive Pipelining**: Introduced a k-step lookahead methodology within each PE, allowing for intensive pipelining and eliminating data dependencies that hinder pipeline performance. This approach enables each PE to operate at maximum frequency, contributing to an overall reduction of up to **37%** in PPO training time after eliminating the delay associated with GAE computation phase.

- **Design of an Efficient Memory Layout System**: Created a system that organizes rewards, values, advantages, and rewards-to-go on-chip using dual-ported Block RAM (BRAM) with a First-In Last-Out (FILO) storage mechanism. This design provides the required throughput each cycle, reduces memory access latency, and allows overwriting of the same memory locations for efficient data handling. This was found to be taking alone up to 11% of the training time in CPU-GPU systems according to the profiling in Table 4.2.

- **Integration of the PPO Pipeline on a Single System-on-Chip (SoC)**: Enabled the integration of multiple custom hardware components, memory, and CPU cores on a single SoC architecture, accommodating all phases of PPO from environment simulation to GAE computation. This integration reduces communication overhead, enhances data throughput, and improves overall system performance.

Algorithm Analysis

- **In-Depth Time Profiling Analysis**: Conducted a detailed analysis to identify computational bottlenecks within the PPO algorithm. The findings revealed that the whole GAE phase accounts for up to **48.6%** of processing time in CPU-GPU systems, underscoring the necessity of optimizing this phase to improve overall performance.

6.2 Conclusion

This thesis addresses critical challenges in the efficient implementation of deep reinforcement learning algorithms for real-time applications on resource-constrained hardware platforms, with a particular focus on the Proximal Policy Optimization (PPO) algorithm. Key innovations include the introduction of dynamic reward standardization, which enhanced training stability and performance across environments, yielding up to a **67% increase in cumulative rewards**. By implementing carefully selected standardization and quantization techniques—most notably an 8-bit uniform quantization strategy—this work achieved a **75% reduction in memory requirements** without compromising model performance, ensuring efficient usage of on-chip memory resources.

The development of the HEPPO microarchitecture, with its highly parallelized design and the implementation of a k-step lookahead approach, significantly accelerates the GAE computation phase. HEPPO's ability to process **19.2 billion elements per second** using 64 PEs, with each PE handling **300 million elements per second**, contributes to an overall reduction in PPO training time that goes up to **37%** in some environments.

The efficient memory layout system further enhances the performance of the hardware implementation by optimizing data access patterns and reducing memory latency, pushing the overall reduction to **48%**. The in-depth time profiling analysis provided valuable insights into the computational bottlenecks of the PPO algorithm, guiding the optimization efforts focused on the GAE computation phase.

By integrating the entire PPO pipeline on a single SoC, we have demonstrated a viable path toward deploying reinforcement learning algorithms in embedded systems

and real-time applications. This integration reduces communication overhead and leverages the advantages of custom hardware acceleration to enhance data throughput and overall system performance.

Collectively, these contributions bridge the gap between advanced reinforcement learning algorithms and practical hardware implementations, paving the way for future developments in high-performance, efficient reinforcement learning systems.

6.3 Future Work

Building on the foundations established in this thesis, several avenues for future research are identified.

6.3.1 High-Level Synthesis for Environment Acceleration

One significant bottleneck in reinforcement learning applications is environment simulation, especially in tasks involving complex physics or high-dimensional state spaces. High-Level Synthesis (HLS) offers a promising approach to accelerating environment simulations by translating high-level code into hardware descriptions. Future work could explore the implementation of environment simulations on hardware using HLS tools. By accelerating the environment alongside the PPO algorithm, the entire reinforcement learning loop can be optimized, further reducing training times and enabling more complex or real-time applications.

6.3.2 Complete SoC Implementation and Experimental Evaluation

While the design space exploration of integrating the full PPO pipeline on an SoC has been initiated, a complete implementation and experimental evaluation remain as future work. This involves:

- **Hardware Implementation**: Developing and synthesizing the complete hardware design, including the HEPPO accelerator, neural network accelerators, and environment simulation components.
- **Hardware-Software Co-Design**: Creating a cohesive framework that integrates software components with the hardware accelerators, ensuring efficient communication and synchronization.
- **Experimental Evaluation**: Benchmarking the full system on various reinforcement learning tasks to assess performance gains, resource utilization, and energy efficiency.
- **Scalability Analysis**: Investigating how the architecture scales with more complex environments or larger neural network models, identifying potential bottlenecks and optimization opportunities.

6.3.3 Adaptive Precision and Approximate Computing

Further optimization of hardware efficiency can be achieved by exploring adaptive precision techniques and approximate computing. By adjusting the precision of computations based on the sensitivity of different parts of the algorithm, it is possible

to reduce resource usage without significantly impacting performance. Research can focus on:

- **Dynamic Precision Scaling**: Implementing mechanisms that adjust the bit-width of computations in real-time, depending on the requirements of different algorithm phases.
- **Error Tolerance Analysis**: Studying the impact of approximation on learning outcomes to determine acceptable trade-offs between accuracy and efficiency.
- **Energy Efficiency Improvements**: Evaluating how adaptive precision contributes to lower power consumption, which is critical for embedded and portable applications.

6.3.4 Extending Algorithmic Modifications to Other Algorithms

The algorithmic modifications proposed, such as dynamic standardization and quantization techniques, could be extended to other reinforcement learning algorithms. Investigating their applicability and effectiveness in algorithms like Deep Q-Networks (DQN) or Soft Actor-Critic (SAC) can broaden the impact of this research and contribute to the development of hardware-efficient solutions for a wider range of reinforcement learning algorithms.

6.3.5 Tool Development and Framework Integration

Developing tools and frameworks to facilitate the design and deployment of hardware-accelerated reinforcement learning systems can lower the barrier to entry for practitioners. Integrating the proposed solutions into widely used platforms, such as TensorFlow or PyTorch, with hardware acceleration support, can promote adoption and further innovation.

6.4 Closing Remarks

The advancements presented in this thesis contribute significantly to bridging the gap between sophisticated reinforcement learning algorithms and efficient hardware implementations. By addressing key computational bottlenecks and proposing scalable, hardware-friendly solutions, this work lays the groundwork for deploying advanced reinforcement learning systems in practical, real-world applications. The combination of algorithmic innovations and hardware acceleration techniques holds the promise of unlocking new capabilities in autonomous systems, robotics, and embedded intelligence. Continued research and development in this area are essential for realizing the full potential of reinforcement learning in diverse and impactful domains.

Bibliography

[1] S. Bhatt. Reinforcement learning 101: Learn the essentials of reinforcement learning! *Towards Data Science*, Mar. 2018. URL https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[2] S. Chatterjee. 7 applications of reinforcement learning in finance and trading. https://neptune.ai/blog/7-applications-of-reinforcement-learning-in-finance-and-trading, Sept. 2024. Accessed: 13th September, 2024.

[3] J. Chen, C. Zhao, S. Jiang, X. Zhang, Z. Li, and Y. Du. Safe, efficient, and comfortable autonomous driving based on cooperative vehicle infrastructure system. *International Journal of Environmental Research and Public Health*, 20(1):893, 2023. doi: 10.3390/ijerph20010893.

[4] W. Chen, Z. Zhang, D. Tang, C. Liu, Y. Gui, Q. Nie, and Z. Zhao. Probing an lstm-ppo-based reinforcement learning algorithm to solve dynamic job shop scheduling problem. *Computers & Industrial Engineering*, 197:110633, Nov. 2024. doi: 10.1016/j.cie.2024.110633. URL https://doi.org/10.1016/j.cie.2024.110633.

[5] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman. Phasic policy gradient. *arXiv preprint arXiv:2009.04416*, 2020. URL `https://doi.org/10.48550/arXiv.2009.04416`. Submitted on 9 Sep 2020.

[6] S. Dalton, I. Frosio, and M. Garland. Accelerating reinforcement learning through gpu atari emulation. *arXiv preprint arXiv:1907.08467*, 2019. URL `https://arxiv.org/abs/1907.08467`. Version 2, last revised 5 Oct 2020.

[7] J. Deng, S. Sierla, J. Sun, and V. Vyatkin. Reinforcement learning for industrial process control: A case study in flatness control in steel industry. *Computers in Industry*, 143:103748, Dec. 2022. doi: 10.1016/j.compind.2022.103748.

[8] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. *arXiv preprint arXiv:1604.06778*, 2016. URL `https://doi.org/10.48550/arXiv.1604.06778`. 14 pages, ICML 2016, Version 3, last revised 27 May 2016.

[9] H. Fan, G. Wang, M. Ferianc, X. Niu, and W. Luk. Static block floating-point quantization for convolutional neural networks on fpga. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Dec 2019. doi: 10.1109/ICFPT47387.2019.00012.

[10] F. Foundation. Bipedalwalker-v3 environment. `https://gymnasium.farama.org/environments/box2d/bipedal_walker/`, 2023.

[11] F. Foundation. Cartpole-v1 environment. `https://gymnasium.farama.org/environments/classic_control/cart_pole/`, 2023.

[12] F. Foundation. Halfcheetah-v5 environment. https://gymnasium.farama.org/environments/mujoco/half_cheetah/, 2023.

[13] F. Foundation. Humanoid-v5 environment. https://gymnasium.farama.org/environments/mujoco/humanoid/, 2023.

[14] F. Foundation. Lunarlander-v3 environment. https://gymnasium.farama.org/environments/box2d/lunar_lander/, 2023.

[15] S. Fox, S. Rasoulinezhad, J. Faraone, david boland, and P. Leong. A block minifloat representation for training deep neural networks. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=6zaTwpNSsQ2.

[16] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[17] L. N. Govindarajan, R. G. Liu, D. Linsley, A. K. Ashok, M. Reuter, M. J. Frank, and T. Serre. Diagnosing and exploiting the computational demands of video games for deep reinforcement learning. *arXiv preprint arXiv:2309.13181*, 2023. URL https://arxiv.org/abs/2309.13181.

[18] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *arXiv preprint arXiv:1610.00633*, 2016. URL https://doi.org/10.48550/arXiv.1610.00633. Version 2, last revised 23 Nov 2016.

[19] C. Guo, B. Lou, X. Liu, D. Boland, P. H. Leong, and C. Zhuo. BOOST: Block Minifloat-Based On-Device CNN Training Accelerator with Transfer Learning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, San Francisco, CA, USA, October 28–November 2 2023. IEEE. doi: 10.1109/ICCAD57390.2023.10323638. URL `https://ieeexplore.ieee.org/document/10323638`.

[20] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[21] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[22] M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. *arXiv preprint arXiv:1906.08253*, 2019. URL `https://arxiv.org/abs/1906.08253`. Version 3, last revised 29 Nov 2021.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, United States, 3rd edition, Nov. 1997. ISBN 978-0-201-89684-8.

[24] P. Kormushev, S. Calinon, and D. G. Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013. doi: 10.3390/robotics2030122. URL `https://doi.org/10.3390/robotics2030122`.

[25] N. P. Lawrence. *Deep Reinforcement Learning Agents for Industrial Control System Design.* Phd thesis, The University of British Columbia, Vancouver, 2023. © Nathan P. Lawrence, 2023.

[26] C. R. Lazaridis, I. Michailidis, G. Karatzinis, P. Michailidis, and E. Kosmatopoulos. Evaluating reinforcement learning algorithms in residential energy saving and comfort management. *Energies*, 17(3):581, 2024. doi: 10.3390/en17030581.

[27] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. *arXiv preprint arXiv:1810.05762*, 2018. URL `https://arxiv.org/abs/1810.05762`.

[28] H. H. Loomis and B. Sinha. High-speed recursive digital filter realization. *Circuits, Systems, and Signal Processing*, 3:267–294, 1984.

[29] Z. Mammeri. Reinforcement learning based routing in networks: Review and classification of approaches. *IEEE Access*, 7:55916–55950, Apr 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2913776. Open Access.

[30] H. Mania, A. Guy, and B. Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018. URL `https://arxiv.org/abs/1803.07055`. Submitted on 19 Mar 2018.

[31] mbcel. Understanding normalization of advantage function in ppo. `https://github.com/openai/baselines/issues/544`, 2018.

[32] Y. Meng, S. Kuppannagari, and V. Prasanna. Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*

(FCCM), School of Electrical and Computer Engineering, University of Southern California, 2020. IEEE.

[33] OpenAI. Proximal policy optimization (ppo). `https://spinningup.openai.com/en/latest/algorithms/ppo.html`, 2018. URL `https://spinningup.openai.com/en/latest/algorithms/ppo.html`. Revision 038665d6.

[34] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley, 1999. ISBN 978-0-471-24186-7.

[35] K. K. Parhi and D. G. Messerschmitt. Pipeline interleaving and parallelism in recursive digital filters-part i: Pipelining using scattered look-ahead and decomposition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37: 1099–1117, July 1989.

[36] K. K. Parhi and D. G. Messerschmitt. Pipeline interleaving and parallelism in recursive digital filters-part ii: Pipelined incremental block filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37:1118–1134, July 1989.

[37] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. URL `https://arxiv.org/abs/1506.02438`. Last revised 20 Oct 2018, version 6.

[38] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2017. URL `https://arxiv.org/abs/1502.05477`. Version 5, last revised 20 Apr 2017.

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[40] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. URL `https://doi.org/10.48550/arXiv.1712.01815`.

[41] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017. doi: 10.1038/nature24270.

[42] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[43] A. Tammewar, N. Chaudhari, B. Saini, D. Venkatesh, G. Dharahas, D. Vora, S. Patil, K. Kotecha, and S. Alfarhood. Improving the performance of autonomous driving through deep reinforcement learning. *Sustainability*, 15(18): 13799, 2023. doi: 10.3390/su151813799.

[44] P. Thodoroff, W. Li, and N. D. Lawrence. Benchmarking real-time reinforcement learning. In *Proceedings of Machine Learning Research, NeurIPS 2021 Preregistration Workshop*, volume 181, pages 26–41, 2022.

[45] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The computational

limits of deep learning. 2020. URL https://arxiv.org/abs/2007.05558. Version 2, last revised 27 Jul 2022.

[46] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi: 10.1080/00401706.1962.10490022. URL https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022. Published online: 30 Apr 2012.

[47] J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. Yan. Envpool: A highly parallel reinforcement learning environment execution engine. *arXiv preprint arXiv:2206.10558*, 2022. URL https://arxiv.org/abs/2206.10558.

[48] Z. Yang. Justifying advantage normalization for ppo. https://github.com/DLR-RM/stable-baselines3/issues/485, 2021.

[49] C. Yu, A. Velu, E. Vinitsky, J. Gao, Y. Wang, A. Bayen, and Y. Wu. The surprising effectiveness of ppo in cooperative, multi-agent games. *arXiv preprint arXiv:2103.01955*, 2022. URL https://doi.org/10.48550/arXiv.2103.01955. Accepted by NeurIPS 2022 Datasets and Benchmarks, Version 4, last revised 4 Nov 2022.

[50] F. Zejnullahu, M. Moser, and J. Osterrieder. Applications of reinforcement learning in finance: Trading with a double deep q-network. *arXiv preprint arXiv:2206.14267*, June 2022. URL https://arxiv.org/pdf/2206.14267.

[51] C. Zhang, J. Cheng, I. Shumailov, G. A. Constantinides, and Y. Zhao. Revisiting block-based quantisation: What is important for sub-8-bit llm inference? *arXiv*

preprint arXiv:2310.05079, 2023. URL `https://arxiv.org/abs/2310.05079`. Version 2, last revised 21 Oct 2023.

[52] H. Zhang and T. Yu. Taxonomy of reinforcement learning algorithms. In H. Dong, Z. Ding, and S. Zhang, editors, *Deep Reinforcement Learning*. Springer, Singapore, 2020. ISBN 978-981-15-4094-3. doi: 10.1007/978-981-15-4095-0_3. URL `https://doi.org/10.1007/978-981-15-4095-0_3`.

[53] G. Zhao, J. Xu, A. Liu, and J. Yu. Research on proximal policy optimization algorithm based on n-step update. In *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, Beijing, China, May 2021. IEEE. doi: 10.1109/CISCE52179.2021.9445929. Date Added to IEEE Xplore: 09 June 2021.