Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

DA399E
Spring Term 2024

Independent project (degree project), 15 credits, for the Degree of Bachelor of Science (180 credits) with a major in Computer Science
Spring Semester 2024
Faculty of Natural Sciences

Implementing and Comparing Reinforcement Learning Algorithms in Embedded and Traditional Computing Environments for Autonomous Systems

Adis Veletanlic, Dzenis Madzovic

May 28, 2024

Authors
Adis Veletanlic, Dzenis Madzovic

Title
Implementing and Comparing Reinforcement Learning Algorithms in Embedded and Traditional Computing Environments for Autonomous Systems

Supervisor
Dawit Mengistu

Examiner
Niklas Gador

Abstract
This study explores the application and comparative performance of reinforcement learning (RL) algorithms in embedded systems versus traditional computing environments, focusing on autonomous systems. We implement and evaluate Q-Learning and Deep Q-Learning across diverse hardware setups, including the NVIDIA Jetson Nano and high-power desktop computers. Our methodology involves training these algorithms to maintain equilibrium in simulated environments, with an emphasis on computational efficiency and resource constraints in embedded systems.

The results demonstrate that while traditional environments offer faster computation, embedded systems can achieve comparable accuracy in specific tasks, albeit at a slower pace, and also do this at higher energy efficiency. This research highlights the trade-offs between execution time and performance across different computing platforms. We also explore software optimizations, like using Numba for Q-Learning, which significantly reduces execution time on embedded systems. The findings suggest practical applications and potential optimizations for RL in environments where computational resources are limited, paving the way for more efficient autonomous systems in real-world applications.

Keywords
Reinforcement Learning, Embedded Systems, Autonomous Systems, Machine Learning

Table of Contents

1 Introduction **1**

1.1 Problem Definition 1

1.2 Method 2

1.3 Motivation 2

1.4 Research Question 3

2 Background **4**

2.1 Autonomous Robotic Systems 4

2.2 Q-Learning and Deep Q-Learning 4

2.3 Literature Review 5

2.4 Proposed Optimization Techniques 6

2.5 Practical Implementations of RL on Embedded Systems 8

2.6 Performance Metrics and Evaluation 8

3 Implementation **10**

3.1 Training Enviroment and Hardware 10

3.2 Simulated Environment and Physical Benchmarking 10

3.2.1 Implementation of the Simulated Environment 11

3.3 Implementation of Q-Learning 12

3.3.1 State Discretization 13

3.3.2 Action Selection Strategy 14

3.3.3 Updating the Q-table 15

3.4 Software Optimizations for the Q-Learning implementation 16

3.5 Implementation of Deep Q-Learning 16

3.5.1 Implementation of Deep Q-Learning with fixed Q-Value Targets . . 18

3.5.2 Loss Function for DQL Implementations and Differences 18

4 Experiment **19**

4.1 Experiment Setup . . . 19

4.2 Implementation Evaluation . . . 20

4.2.1 Reward Analysis . . . 20

4.2.2 Custom Visualization . . . 20

4.3 Performance Metrics . . . 21

4.3.1 Metrics Collection on The Embedded System . . . 21

4.3.2 Metrics Collection on The Desktop Environment . . . 21

5 Results and Discussion **22**

5.1 Reward Plots for Classic Q-Learning and Optimized Q-Learning Using Numba . . . 22

5.2 Reward Plots for Deep Q-Learning and Deep Q-Learning with fixed target values . . . 23

5.3 Benchmark Metrics . . . 24

5.3.1 Jetson Nano . . . 24

5.3.2 Average Benchmark Metrics . . . 25

5.3.3 Machine Learning Server . . . 28

5.3.4 Average Benchmark Metrics . . . 28

5.4 Performance per Watt for Embedded System and Machine Learning Server 31

5.5 Execution Times for Embedded System and Machine Learning Server . . . 32

6 Conclusion **35**

1. Introduction

The advancement of autonomous systems is transforming how we interact with technology, reshaping industries, and redefining efficiency in our daily lives. As these systems gain complexity and are expected to perform tasks with higher levels of autonomy, the role of efficient machine learning algorithms, particularly in the domain of **reinforcement learning** (RL), becomes increasingly pivotal. RL stands at the forefront of the revolution in autonomy, providing the foundation for machines to learn from interactions and make decisions that mirror human judgment, often surpassing it in precision and reliability [1].

Our thesis explores the implementation and comparative analysis of RL algorithms within two distinct computational paradigms – the field of embedded systems and edge computing, and traditional desktop computing environments made for heavy computation. By investigating the performance and resource utilization of these algorithms during the training and operation phases, this research seeks to unveil insights into their adaptability and scalability across varying hardware capabilities, as well as what can be done to **more easily** optimize the algorithms given certain resource constraints, and what the implications of this are.

Our exploration is set against the backdrop of autonomous Q-learning systems and deep Q-learning systems, where the balance between computational resource constraints and the demand for real-time processing poses a unique challenge. The study is tailored to assess how different computing environments influence the efficiency and speed of learning, and ultimately the performance of autonomous control systems. Use cases of such systems are increasingly prevalent, encompassing autonomous vehicles, aerial drones, and robotic systems in various domains. These applications span a spectrum of industries, including but not limited to surveillance [2], agriculture [3], logistics [4], and healthcare [5].

1.1. Problem Definition

Our research aims to address the challenge of applying Q-Learning and Deep Q-Learning in resource-constrained embedded systems compared to high-power computing environments. We investigate the performance trade-offs, including processing power, energy use, and training time of these reinforcement learning methods when adapted to smaller, less powerful systems. Through benchmarking in scenarios like controlling an inverted pendulum robot, our goal is to uncover strategies that mitigate the limitations of embedded systems in running complex reinforcement learning algorithms. Our efforts will try to enhance the feasibility and efficiency of deploying autonomous systems in environments where computing resources are limited.

1.2. Method

Our method involves comparing Q-Learning algorithm performance on an NVIDIA Jetson Nano (representing embedded systems) against traditional desktop computing setups in training a cart-pole system to maintain equilibrium. This experiment highlights control challenges and evaluates the algorithm's adaptability to resource-constrained environments.

- **Experimental Setup:** The Jetson Nano is chosen for its computational efficiency suitable for embedded scenarios, while a high-power computing environment serves as our benchmark for comparison. The primary task involves the cart-pole, a classic testbed for illustrating non-linear control dynamics.

- **Algorithm Implementation:** We implement Q-Learning, favored for its low computational demand and effectiveness in discrete action spaces, across both environments, and Deep Q-Learning. This involves tailoring the algorithms for optimal performance within the embedded system's constraints.

- **Performance Evaluation:** Key metrics, such as accuracy, energy consumption, temperature, and efficiency, will be measured during training and operation phases. This evaluation helps quantify the trade-offs in deploying reinforcement learning in embedded systems.

- **Data Collection and Analysis:** Performance data from both setups will be analyzed to compare the efficacy and practicality of Q-Learning and Deep Q-Learning in embedded versus resource abundant computing environments, as this analysis aims to understand the algorithm's scalability and optimization needs for resource-limited systems.

- **On-device Training Consideration:** A significant aspect of our methodology involves exploring the potential for on-device training within the embedded environment. This approach, if viable, could significantly enhance the autonomy and efficiency of embedded autonomous systems by reducing the need for data transmission to more powerful external computing resources.

1.3. Motivation

Our project investigates the potential of Q-Learning, a reinforcement learning technique, for controlling an inverted pendulum (cart-pole) system in resource-constrained embedded environments, compared to traditional machine learning computing setups. Highlighting the importance of efficient resource management, we aim to explore how Q-Learning can manage complex tasks with minimal resources while maintaining accuracy, using the cart-

pole—a benchmark experiment for non-linear control—as our testbed. Our study is motivated by the transformative impact of reinforcement learning on autonomous systems, especially in navigation and adaptation within dynamic environments, as demonstrated by studies like [6] and [7].

We will evaluate the performance and resource consumption of Q-Learning on an efficient embedded platform, the OKdo Nano C100, against a more resource-abundant computing environment. Our research focuses on assessing the trade-offs in performance metrics like response time, accuracy, and energy usage, aiming to establish whether Q-Learning can achieve comparable effectiveness in embedded systems as it does in larger computing environments.

By demonstrating the practicality of Q-Learning for real-time or on-device training in embedded systems, our work seeks to advance autonomous system development for scenarios where computing resources are limited. This could lead to more energy-efficient, cost-effective autonomous solutions across various applications, addressing the challenges of speed, efficiency, and power consumption critical to edge-computing devices.

1.4. Research Question

How do Q-Learning algorithms perform in terms of computational efficiency, accuracy, and energy consumption on resource-constrained embedded systems, and what are the implications of these factors on the real-world applicability of training and operating autonomous systems?

2. Background

2.1. Autonomous Robotic Systems

An autonomous robot, or autonomous system, is a system which can operate without the need for human control. Under the course of history, especially in the effort to control systems to achieve autonomy, practices such as PID control (Proportional-Integral-Derivative) have been used extensively in such control engineering tasks. The goal of such algorithms is to manually determine the best course of action for a system depending on some state, given by its environment, with which it can then mathematically determine what to do. More efforts are actively being made to automate the **learning** itself – as opposed to just the controlling behavior of the system – using machine learning, specifically reinforcement learning.

2.2. Q-Learning and Deep Q-Learning

Q-Learning, introduced by Watkins in 1989 [8], stands as a foundational technique in model-free reinforcement learning, enabling agents to learn optimal actions in Markovian domains through direct interaction with the environment. This method, relying on the *Q-Matrix*, allows an agent to learn without constructing explicit maps of the domain, making decisions based on immediate rewards or penalties and the estimated future value of resulting states. Operating in a discrete action space, Q-Learning systematically explores all actions across states to incrementally identify the most beneficial actions through a reward function $S \times A \to \mathbb{R}$, or $Q(s, a)$, i.e. the set of all possible ordered pairs (s, a), where $s \in S$ and $a \in A$.

Building on the principles of Q-Learning, Deep Q-Learning (DQL) integrates deep learning to extend the method's applicability to complex, high-dimensional environments that are challenging for traditional Q-Learning to handle. Introduced as a significant advancement by Mnih et al. in 2013 [9], Deep Q-Learning utilizes deep neural networks (DNNs) as function approximators to represent the Q-value function. This approach enables the algorithm to learn optimal policies over **continuous** state spaces (not necessarily present in our practical experiments), addressing the scalability issues faced by traditional Q-Learning when dealing with large or continuous action spaces.

Deep Q-Learning's key innovation lies in its ability to generalize over similar states, reducing the need for the algorithm to experience every possible state-action pair to learn effectively. This is achieved through the use of a technique known as experience replay, where transitions are stored in a replay buffer and sampled randomly to update the network. This process breaks the correlation between consecutive learning samples, leading to more stable and efficient learning. Furthermore, DQL often employs separate networks for generating the target Q-values during the training process – a technique known as

target network freezing – to enhance learning stability.

The integration of deep neural networks into Q-Learning facilitates the application of reinforcement learning in a broader range of real-world tasks, such as autonomous driving, game playing, and robotics, where the environment's complexity and the high dimensionality of the state space make traditional methods less effective. By leveraging the representational power of deep learning, Deep Q-Learning not only enhances the agent's ability to understand and interact with complex environments but also opens up new possibilities for the efficiency and adaptability of autonomous systems in both research and practical applications.

2.3. Literature Review

In preparation for the implementation phase of our research, we conducted an exhaustive literature review to evaluate the current state of research on the application of Reinforcement Learning (RL) algorithms within embedded systems for controlling autonomous systems, and cases where optimization techniques have been deployed to further on-device training and the possible implications and benefits of doing this. This review was aimed at understanding RL's performance, efficiency, and adaptability in the context of embedded systems, which are often resource-constrained environments. The literature review results also empirically highlighted that the most popular and commonly implemented family of machine learning for non-linear autonomous systems was Q-Learning.

This literature review laid the groundwork for our research by pinpointing the gaps in existing knowledge and highlighting areas for potential investigation. The insights gleaned from this comprehensive review have been instrumental in informing the design and methodology of our study. Specifically, they have shaped our approach to implementing and evaluating RL algorithms within embedded systems, with a focus on optimizing performance and practicality for autonomous control applications.

- **Proposed Optimization Techniques:** We examined studies that have explored the feasibility and performance of RL algorithms on low-power devices, including microcontrollers, embedded systems, and edge devices, which focused primarily on hardware optimizations or distributed approaches for embedded systems communicating over a shared network. This involved a critical evaluation of the specific adaptations or modifications to traditional RL algorithms that enable them to operate effectively under the constraints of limited computational power, memory, and energy availability [10] [11]. Other studies focused on theoretical and applied optimization techniques in low-power systems such as embedded systems and mobile devices [12] [13].

  Another study highlights the adaptability and possibility of on-device machine

learning for embedded systems [14]. This study focused on techniques based on neural networks and transfer learning, which are less common in Reinforcement Learning, but still apply to our study as we will later delve into the practical implementation of Deep Q-Learning, a technique that is potentially more power intensive but possibly better in regards to time and scope of training. We are also interested in the trade-offs and potential of complete on-device training of Q-Learning algorithms for embedded systems, with or without transfer learning.

- **Practical Implementations of RL on Embedded Systems:** Here we review some practical case studies where RL has been effectively applied within embedded systems for autonomous control [15] [16]. These cases provided insights into the real-world applicability of RL algorithms in scenarios where power efficiency and processing capability are critical factors, as well as possible avenues for better representing the challenges and opportunities in the deployment of RL within constrained environments. A study on *"Formalism and Benchmarking"* [17] highlights RL's effectiveness in autonomous settings without human intervention, emphasizing its adaptability in dynamic conditions. It discusses designing and benchmarking RL algorithms for non-episodic training, crucial for embedded systems requiring **continuous adaptation without resets**. This method is vital for systems needing long-term, independent control, suggesting that autonomous RL could greatly improve embedded systems' performance in applications like industrial automation and smart infrastructure.

- **Performance Metrics and Evaluation:** Finally, our literature review assessed the metrics commonly employed to evaluate the efficacy of RL algorithms in embedded systems. We paid special attention to how metrics that are particularly relevant to embedded contexts are measured, such as energy consumption (per decision), computational load, and algorithmic convergence rates [11].

2.4. Proposed Optimization Techniques

This is still a novel area, but some previous efforts have been made to uncover potential optimization techniques for higher-level programming languages using machine learning or reinforcement learning libraries such as PyTorch and Tensorflow, substituting these for other more lightweight libraries. A notable approach involves substituting these standard libraries with more lightweight alternatives, aiming to alleviate the computational burden on constrained devices. Among the array of innovative techniques, a few have emerged as particularly promising

- **Binarization of Weights:** This simplifies the model by converting weights to binary values, significantly reducing memory requirements and computational com-

plexity.

- **Pre-compilation to Lower-level Languages:** By translating models into lower-level languages, we can optimize execution efficiency, capitalizing on the strengths of each device's hardware.

- **Conversion of Floating-point Numbers to Fixed-point:** This technique adjusts the numerical precision of model parameters, offering a balance between performance and resource consumption.

- **Pruning:** By systematically removing less important connections or weights within the network, pruning helps in streamlining the model, further enhancing its suitability for resource-constrained environments.

However, the majority of the efforts made in this research is transferring already pre-trained models or weights from one, less resource constrained, environment to one that has severe limitations in terms of processing and GPU power [14]. As previously suggested by F. Svoboda et al. [12], with regards to Deep reinforcement learning on resource constrained devices:

> It is unclear for example, how well these algorithms would perform when potential avenues of optimization are explored – and if acceptable control and decision performance could be maintained within the constraints of tinyML devices.

A critical question could be how these techniques generalize across different RL algorithms beyond Deep Q-Networks (DQN), including those not covered in the paper. Can these optimizations be effectively applied to a wide range of RL algorithms used in autonomous systems, and what are the performance trade-offs? We will not be delving into this in our thesis necessarily, but it is important to question nonetheless, as we will be primarily focusing on classical Q-Learning and on-device training.

It would be interesting to explore how these optimizations impact the learning dynamics and convergence rates of the RL algorithms. For example, does the reduction in precision or the minimization of redundancy affect the stability or speed of learning, and **how does this vary between embedded and traditional computing environments**?

2.5. Practical Implementations of RL on Embedded Systems

There are many benefits to reinforcement learning on embedded systems. Data and processing is near the source, enhanced privacy and security, increased energy efficiency, and cost-effectiveness. Reducing dependence on things like cloud services can lower operational costs related to data storage and processing. For businesses deploying large numbers of IoT devices, these savings can be substantial. Most edge computing systems utilizing reinforcement learning on embedded devices deploy special techniques to cope with the resource constraints of the systems, such as policy distillation techniques [18]. Other papers highlight the difficulty of deploying reinforcement learning algorithms on autonomous vehicles in particular, and the implications and necessary human effort of employing such strategies [16].

2.6. Performance Metrics and Evaluation

As previously mentioned, to properly and accurately measure and evaluate the efficacy of reinforcement learning (RL) algorithms within embedded systems, an approach that considers both the inherent and superficial metrics relevant to the operational environment of these systems is crucial. These metrics provide a view of an algorithm's performance, factoring in not only its computational efficacy but also its practical viability in resource-constrained scenarios.

- **Energy Consumption:** One of the most important considerations in embedded systems is energy efficiency. The energy consumed, per decision and on average, made by the RL algorithm is a critical metric, especially in battery-powered or energy-harvesting devices where power is limited. Measurement of energy consumption can be achieved through direct electrical measurements of the device during algorithm operation, allowing for an assessment of the algorithm's energy efficiency relative to its decision-making capabilities.

- **Computational Load:** This is evaluated by the amount of computational resources the RL algorithm requires during operation, including CPU usage and memory. Tools such as profilers and memory trackers are utilized to gauge these resources, providing insights into the algorithm's complexity and its suitability for low-power and low-memory embedded devices. This metric is important for understanding the trade-offs between algorithm complexity and system resource constraints, as well as properly determining if any practical optimization efforts are worth the work.

- **Practical Evaluation Methodologies:** To properly assess these metrics, a combination of simulation-based evaluations and real-world empirical testing is often employed. A simulation allows for controlled experimentation and quick iteration

over algorithms, while empirical testing on actual embedded hardware provides validation of the algorithm's performance in practical scenarios. Such methodologies enable a more complete understanding of an RL algorithm's behavior under the specific constraints and conditions of embedded systems.

3. Implementation

3.1. Training Enviroment and Hardware

The training environment consists of a custom cart pole implementation, or inverted pendulum, specifically built with a real-life replica in mind [19]. The robot has two encoders, one for determining the position of the cart and another for determining the angle of the pendulum. From these, both angular and positional velocity can be derived, providing yet two additional states. The purpose of a custom implementation is to determine both the feasibility of transfer learning to a physical system and the possible setbacks or implications of this in future research.

The embedded device used for testing in our work is the OKdo Nano C100 Developer Kit, based on the NVIDIA Jetson Nano Developer Kit. It houses a 128-core Maxwell GPU, Quad-core ARM A57 CPU, and 4 GB of 64-bit LPDDR4 RAM. It comes with extensive GPIO, allowing for further testing and implementation on physical hardware devices.

Table 1. OKdo Nano C100 Developer Kit

Component	Specification
CPU	ARM A57 CPU
GPU	Tegra X1 128-core Maxwell GPU
RAM	4GB LPDDR4
Storage	128GB Micro SD + 16GB eMMC
Power Supply	5V/4A DC power adapter
Cooling System	Heatsink (Passive)
Operating System	Ubuntu 18.04 LTS

The more powerful contestant is a Machine Learning server with extensive GPU capabilities and cooling.

Table 2. Machine Learning Server

Component	Specification
CPU	AMD Ryzen 7 7800X3D
GPU	NVIDIA GeForce RTX 4070Ti, 12GiB GDDR6X, 641 AI TOPS
RAM	32GB DDR4
Storage	3TB NVME Drive
Power Supply	Corsair RM1000x Shift 1000W
Cooling System	CPU Heatsink + 3 active cooling fans
Operating System	Ubuntu 22.04 LTS

3.2. Simulated Environment and Physical Benchmarking

As previously stated, the training environment (simulation) used in our experiment was custom-built to replicate a real-life inverted pendulum robot. The cart pole problem itself

is a common task in the realm of reinforcement learning, and a pre-made simulation is already available with the OpenAI Gym toolkit. We decided not to use that simulated environment since the provided values of the simulation were not configurable.

To implement a simulation that produces an agent that can be run directly on the physical robot, firstly we collected relevant measurements from the physical replica. These measurements were the length of the pole, the mass of the pole, the mass of the pendulum (additional weight), the track length, and the maximum and minimum angle of the pendulum. The robot used an M500 Series Industrial DC Servo Motor, which operates at a maximum voltage of $60V$. By applying that voltage to the robot, moving the cart from one side of the track to the other, and measuring the time, we approximated the maximum cart velocity and angular velocity, which we used later on in our training and environment visualizer.

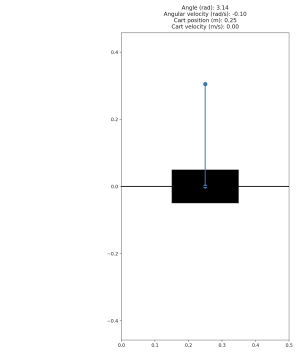

Figure 1. Environment Visualizer

Parameter	Value
Pole Mass (m_{pole})	0.17 kg
Pole Length (L)	0.305 m
Motor's Max Force (F_{max})	2.3 N
Max Deviation from π (θ_{max})	25 rad
Track Length (L_{track})	0.5 m
Pendulum Mass (m_{pendulum})	0.09 kg

Table 3. System Parameters

3.2.1. Implementation of the Simulated Environment

To implement our simulated environment we used OpenAi's "Cart-Pole v1" as a reference on how our environment should work. The implementation had four variables within it's state, $(\theta, \omega, x, \dot{x})$, where:

- θ is the angle of the pendulum,
- ω is the angular velocity of the pendulum,
- x is the cart position, and
- $\dot{x}$ is the cart velocity.

These states are represented internally as the vector,

$$s = \begin{bmatrix} \theta \\ \omega \\ x \\ \dot{x} \end{bmatrix}$$

A method called "*simulate_step*" was implemented that would take an input of either 0 or 1 and calculate the new state of the environment, the reward obtained from the current step, and a boolean flag indicating whether the current state is terminal.

The method calculates the force to be applied to the pendulum based on the given action. This force is determined by the "*_calculate_force*" method, which returns the maximum force (in either the positive or negative direction) depending on the action provided. Using the "*solve_ivp*" function from SciPy[20], the method integrates the equations of motion for the pendulum over a small time interval dt, which is set to $0.02s$. It utilizes the "*equations_of_motion*" method to define the differential equations that govern the nonlinear system dynamics. These equations are solved numerically using a "Runge-Kutta" method (method "RK45" in SciPy).

After integration, the new state of the pendulum (consisting of the pendulum angle, angular velocity, cart position, and cart velocity) is obtained from the solution of the differential equations. The method then enforces constraints on the state using the "*enforce_constraints*" method to ensure that the pendulum remains within valid physical bounds, such as the track boundaries and maximum angle deviation.

The reward is calculated based on the angular position of the pendulum relative to a target angle, which is π radians. The method "*calculate_reward*" computes the absolute difference between the current angle and the target angle. The reward is inversely proportional to this angle difference, with the formula being:

$$reward = \frac{1}{1+\theta}, \quad \theta \neq -1$$

Consequently, the closer the pendulum is to the upright position, the higher the reward it receives. This reward scheme incentivizes the agent to stabilize the pendulum near the desired upright position, aligning to maintain balance throughout the learning process.

The *terminal_state* method determines if the pendulum's current state signifies the end of a training episode. By evaluating if the pendulum exceeds predefined angle or position limits, it returns True for a terminal state or False otherwise, guiding the learning process by providing clear episode endpoints and enabling effective response to termination conditions.

3.3. Implementation of Q-Learning

In general, reinforcement learning is a field of unsupervised learning that has a long history within the realm of machine learning, being used in machine control [21] and video games particularly. As such, there are large amounts of approaches to reinforcement learn-

ing, for our experiment we decided to use Q-Learning and Deep Q-Learning as our two main approaches. Q-Learning is categorized as a model-free algorithm, the agent initially performs actions randomly within the observed environment and gradually improves its performance by exploiting prior knowledge.

First, we devised a basic training loop for the Q-Learning algorithm, that would produce a Q-table. For the environment, we used the aforementioned inverted pendulum simulation and the following general algorithm:

1. Observe the current environment state s, where $s = (\theta, \omega, x, \dot{x})$.
2. Choose an action a, where $a \in \{0, 1\}$, using an exploration strategy (e.g., epsilon-greedy).
3. Perform the chosen action a.
4. Observe the reward r and the new state s'.
5. Update the Q-table for the current state-action pair, with Q-value, using the Q-learning update rule (with hyper-parameters α and γ)
6. Set the current state s to the new state s'.
7. Repeat until the environment is within the terminal state or a specified number of iterations.

Our implementation of the Q-Learning algorithm was written in Python, using the Python package NumPy [22] for calculations and array arithmetics.

3.3.1. State Discretization

In reinforcement learning, the environment's state space can be continuous, encompassing a range of real-valued variables. Given the computational and memory constraints inherent in practical applications, it becomes necessary to discretize this continuous state space into a finite set of discrete states. This process facilitates the implementation of tabular solution methods, such as Q-learning, which operate on discrete state-action pairs.

To achieve state discretization, we first define the bounds for each continuous variable within the state vector $s = (\theta, \omega, x, \dot{x})$, representing the angle, angular velocity, cart position, and cart velocity, respectively. The bounds are determined based on empirical observation and domain knowledge, ensuring they encapsulate the expected range of each variable within the simulated environment.

The bounds for each variable are defined as follows:

$$\textbf{Lower Bounds } (\mathbf{s}_{\min}): \quad \begin{bmatrix} 2.71 \\ 6 \\ 0.0 \\ -1.8 \end{bmatrix}$$

$$\textbf{Upper Bounds } (\mathbf{s}_{\max}): \quad \begin{bmatrix} 3.58 \\ -6 \\ 0.5 \\ 1.8 \end{bmatrix}$$

Given the defined bounds, the discretization process divides the range of each variable into a predetermined number of intervals, referred to as *buckets*. For instance, with 30 buckets allocated for each variable, the continuous range is segmented into 30 discrete intervals. A continuous value is thus mapped to its corresponding bucket, transforming it into a discrete value. This bucketing approach enables the approximation of the continuous state space with a finite, albeit coarse, representation.

3.3.2. Action Selection Strategy

The selection of actions within the context of Q-learning is crucial for balancing exploration with exploitation, a fundamental challenge in reinforcement learning. Our approach utilizes the ϵ-greedy strategy, which incorporates a parameter ϵ to manage the exploration-exploitation trade-off. The ϵ-greedy policy ensures that with probability ϵ, an action is chosen at random (exploration), while with probability $1 - \epsilon$, the action with the highest estimated Q-value for the current state is selected (exploitation).

Given

- episode_index is the current episode index.
- number_episodes is the total number of episodes.
- ϵ is the exploration-exploitation trade-off parameter.
- s is the current state.
- Q is the Q-table containing all Q-values.

our ϵ-greedy action selection strategy is formalized as a piecewise function:

$$a = \begin{cases} \text{random action from} \{0, 1\}, & \text{if episode\_index} < \frac{\text{number\_episodes}}{4} \\ \text{random action from } \{0, 1\}, & \text{with probability } \epsilon \\ \arg\max_a Q(s, a), & \text{with probability } 1 - \epsilon \end{cases}$$

here the probability ϵ is updated according to:

$$\epsilon = \begin{cases} \epsilon \times 0.999, & \text{if episode\_index} > \frac{\text{number\_episodes}}{2} \\ \epsilon, & \text{otherwise} \end{cases}$$

In the action selection process, a random action is chosen with probability ϵ, and the best-known action according to the Q-table (the action with the highest Q-value) is chosen with probability $1 - \epsilon$. The value of ϵ decays by a factor of 0.999 after the halfway point of the total episodes, promoting more exploitation over exploration as the agent becomes more experienced.

3.3.3. Updating the Q-table

To store the Q-values in memory during training, a Q-table is used. A Q-table in our implementation is represented by a multi-dimensional array. It consist of three columns, a column containing the discretized states, and two columns containing the Q-values for the two possible actions.

Q-Table		
State	Action 1	Action 2
$(\theta_1, \omega_1, x_1, \dot{x_1})$	0.75	0.60
$(\theta_2, \omega_2, x_2, \dot{x_1})$	0.90	0.50
$(\theta_3, \omega_3, x_3, \dot{x_1})$	0.65	0.80
...	...	...
$(\theta_n, \omega_i, x_j, \dot{x_k})$	x	y

Table 4. Example of a Q-table implementation.

The Q-learning algorithm updates the Q values based on the equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$$

where

- s is the current state.
- a is the chosen action.
- s' is the new state after taking action a.
- r is the reward received after taking action a in state s.
- γ is the discount factor.
- α is the learning rate.
- $max_{a'} Q(s', a')$ is the max Q-value for the new state s' across all possible actions a'.

We further modify this update function by incorporating a terminal state parameter, t, as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') \cdot (1-t) - Q(s,a)]$$

The term $(1-t)$ nullifies the future reward component $\gamma max_{a'} Q(s',a')$, for example when the cart-pole hits the end of the track, discouraging such behaviour in future training.

3.4. Software Optimizations for the Q-Learning implementation

To be able to train the agent directly on an embedded system in a reasonable amount of time, the implemented code needed to be optimized. The baseline algorithm, including all of the hyper-parameters, shall remain the same to have fair comparable results of training between the un-optimized and optimized code. To ensure that all algorithms achieve the same rewards in the same amount of iterations a default random seed was set before the training training.

To optimize the implementation, the code was first stripped of any possible abstractions that could slow down the execution time. This means that the code was rewritten to only utilize low-level data structures available in Python. Second, all possible I/O operations such as print statements were removed during the training. To validate our assumption that such operations degraded the performance of our implementation, we ran our original code multiple times with and without any print statements during training. After this, we concluded that print statements significantly degraded performance and decided to remove them entirely during training.

These optimizations provided better performance results, but we felt that these results were still not satisfactory to be able to train the agent on an embedded system. Python can be a bottleneck in performance due to its interpreted nature and dynamic typing. To mitigate this and further decrease execution time and resource consumption a just-in-time (JIT) compiler, Numba [23], was used to compile our Python code into native optimized machine code.

3.5. Implementation of Deep Q-Learning

Deep Q-learning (DQL) differentiates from Q-learning by not utilizing a Q-table to store action rewards in memory. Because of this the state of the environment also doesn't have to be discrete. Instead, the state is sent into a neural network which predicts Q-values.

A major hurdle in our project is balancing performance with computational efficiency on the embedded system. Neural networks are adept at capturing the nonlinearity in systems. Thus, our neural network's baseline should start with a model that approximates the behavior of the autonomous system using a linear operation. This linear model acts as

an initial simplification, which we can use to understand the minimum complexity required for the task before considering more computationally intensive non-linear models. This would entail only one **linear** activation layer for the neural network:

$$\begin{cases} \text{Output layer} \rightarrow \text{Dense}(2, \text{activation} = \text{linear}) \end{cases}$$

The neural network architecture (NNA) we ended up using for this problem was chosen after many trial and error attempts at training the agent. We wanted results that could parallel the performance and training quality of the prior Q-learning solutions, while having a reasonable execution time. Different neural network architectures would either produce sub-par training quality results while having a short execution time, or produce high quality results with an execution time of hundreds of hours long on desktop environments. After experimentation, the NNA that provided us with a good balance of training quality and execution time was the following:

$$\begin{cases} \text{Input layer} \rightarrow \text{Dense}(128, \text{activation} = \text{elu}, \text{input\_dim} = 4) \\ \text{Hidden layer} \rightarrow \text{Dense}(64, \text{activation} = \text{elu}) \\ \text{Output layer} \rightarrow \text{Dense}(2, \text{activation} = \text{linear}) \end{cases}$$

In our implementations of DQL we used the same ϵ-greedy approach to choosing actions within our environment as in the Q-learning implementation. But, instead of exploiting the Q-table when the epsilon value gets decreased we use the neural network to predict the next action. To increase the training efficiency, we use an experience buffer solution to store experiences in memory that are randomly sampled during training and used to aid in training the network. The experience buffer, denoted by $E = \{e_1, e_2, \ldots, e_n\}$, consists of tuples $e_i = (s, a, r, s', t)$, where s is the current state, a is the action taken, r is the reward received, s' is the next state, and t is the termination signal. These tuples are sampled from E to perform updates on the network.

During the training process, each episode an action is selected and the reward for that action is calculated. Then the values of that episode get stored in the experiences buffer, the training of the network doesn't occur until there are enough samples to form a batch of experiences, the batch size is defined as a hyper-parameter. Once enough experiences are acquired from past episodes, for each experience in the batch, a Q-value is predicted using the neural network, and a target Q-value is calculated by incorporating the reward and the maximum Q-value for the next state (with discount factor, γ, applied). The neural network is then fitted with the predicted and target Q-values and trained for 100 epochs.

The discount factor, γ, remains the same as in the Q-learning implementation, the

batch size of the experience buffer was set to 100.

3.5.1. Implementation of Deep Q-Learning with fixed Q-Value Targets

DQL with fixed Q-value targets is a variant of DQL where instead of a model predicting values and setting it's own Q-value targets, a "target" model is used to define Q-value targets instead. A target model in this case is just a copy of the "online" model. Fixed Q-value targets helps in reducing training time and mitigating the issues associated with using the same network for both action selection and target calculation. By keeping the target network fixed for a certain period of time and then periodically updating it based on the *update_period* hyper-parameter (in our case every 10 episodes for 100 episodes total), the algorithm can learn more effectively and converge to better policies.

Due to the usage of a target network some changes have to be made within our implementation. During the sampling process of the experiences buffer, another batch gets sampled in order to facilitate the calculation of the target Q-values for each transition sampled from the experiences buffer. This target network calculates the Q-values with the discount factor applied for this batch, instead of the online network. These online network is then again fitted with the predicted and target Q-values and trained for 100 epochs. After a certain number of episodes, decided by the *update_period* hyper-parameter, the target network gets updated with the weights of the online network.

3.5.2. Loss Function for DQL Implementations and Differences

Our two DQL variants share the same simple mean squared error loss function:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$$

Where x_i and y_i are true and predicted Q-values. When it comes to hyper-parameters; the batch size of the experiences buffer remained 100 in both implementation, the discount rate, γ remained 1 in all implementations and the initial ϵ value remained 0.2 in all implementations. Since DQL with fixed Q-value targets achieved better rewards in less episodes during experimentation, the epsilon value was reduced at a lower threshold for that implementation:

$$\epsilon = \begin{cases} \epsilon \times 0.999, & \text{if episode\_index} > \frac{\text{number\_episodes}}{5} \\ \epsilon, & \text{otherwise} \end{cases}$$

4. Experiment

4.1. Experiment Setup

Our experiment involved running three variants of Q-learning algorithms, defined in the Implementation (3.1) section. We will refer to the implementations as Model-less Q-learning (including the same implementation with software optimizations, described in section 3.1), Deep Q-learning, and Deep Q-learning with fixed Q-value targets. These variants were chosen to represent different approaches to Q-learning, each with its advantages and considerations.

To maintain consistency and ensure repeatability of results, all algorithms were run using the same random seed, specifically seed 42. This approach helps eliminate randomness during the exploration phase of the ϵ-greedy algorithm of the implementations and ensures that any observed differences in performance are due to implementation variations rather than random initialization.

The hyper-parameters used for Model-less Q-learning were set as follows:

- Learning rate (α): 1
- Discount factor (γ): 1
- Exploration rate (ϵ): 0.2

These hyper-parameters were chosen based on prior research and trial-and-error to provide a balanced exploration-exploitation trade-off and encourage convergence towards optimal policies.

Each Q-learning variant was trained for a total of 15,000 episodes. This episode count was determined based on pilot experiments and is sufficient to allow the algorithms to learn meaningful strategies and converge toward stable policies within a reasonable time frame.

Details regarding the parameters and setup for Deep Q-learning and Deep Q-learning with fixed Q-value targets were:

- Learning rate (α): 1
- Exploration rate (ϵ): 0.2
- Experience batch sample size: 100 experiences

For the variant with fixed Q-value targets, the target network was updated every 10 episodes. This approach helps stabilize training by reducing the potential for overestimation bias and improving the overall convergence of the Q-values.

The number of episodes varies between the deep Q-learning implementations, with the

regular deep Q-learning (DQL) being trained for 1000 episodes and the deep Q-learning with fixed Q-value targets trained for 100 episodes. During prior experimentation, these episode counts provided comparable training quality, and effectiveness in acquiring optimal policies within the environment was similar, within the same reasonable time frame.

The experiments were conducted on both embedded and desktop systems, utilizing the available hardware resources such as GPUs (using CUDA and TensorFlow) and CPUs. This setup allows us to compare the performance of Q-learning variants across different hardware configurations and assess their scalability and efficiency in resource-constrained environments.

4.2. Implementation Evaluation

The evaluation of our implemented models and algorithms involved several key components, including reward analysis using reward plots and custom visualizations of the system dynamics. These evaluation techniques allowed us to assess the performance, convergence, and behavior of our models comprehensively.

4.2.1. Reward Analysis

To evaluate the learning progress and performance of our reinforcement learning models, we computed and analyzed the cumulative rewards obtained during each episode. The total rewards accumulated over time provide insights into how well the models are learning and optimizing their policies to achieve the desired objectives. We utilized the Python library Matplotlib to generate plots depicting the trend of total rewards across episodes for each variant of the implemented algorithms.

4.2.2. Custom Visualization

In addition to reward analysis, we developed a custom visualizer, specifically tailored to visualize the dynamics of our own inverted pendulum simulation environment. This visualizer generates animated GIF recordings that illustrate the behavior of the pendulum in response to different actions and environmental conditions. The visualizations provide a qualitative understanding of how the models control and stabilize the pendulum over time.

The visualizer allows interactive exploration of the pendulum's movements and displays relevant information such as angle, angular velocity, cart position, and cart velocity during the simulation. The visualizer can also be run interactively, where users can apply actions (left or right) to observe the system's response.

By combining reward analysis with dynamic visualizations, we were able to holistically evaluate the performance and behavior of our reinforcement learning implementa-

tions. These evaluation techniques provide a comprehensive understanding of how well our models learn and adapt to the complex control task of balancing an inverted pendulum, facilitating informed decision-making and further refinement of the algorithms.

4.3. Performance Metrics

The primary focus of our thesis is on the metric collection and comparative analysis of running the implemented models on the Jetson Nano embedded system versus a traditional desktop system. This evaluation aims to assess the performance, computational efficiency, and resource utilization of our reinforcement learning algorithms in different hardware environments.

4.3.1. Metrics Collection on The Embedded System

During the training of our reinforcement learning models on the Jetson Nano embedded system, we utilized "jtop" (a wrapper around the tegrastats utility from NVIDIA) to gather real-time system metrics. These metrics provide valuable insights into the system's performance, resource utilization, and operational characteristics throughout the training process. The following metrics were collected and analyzed on the embedded system: Time, CPU and GPU Utilization, Memory Utilization, CPU and GPU Temperature, CPU and GPU Voltage, Total Voltage and Current, and Average Power Consumption.

By tracking CPU and GPU utilization, we can monitor the system's performance and workload distribution during training. CPU and GPU temperature readings give us insights into thermal conditions, overheating possibilities, and operation stability over extended training sessions. Voltage, current, and power consumption metrics contribute to assessing the energy efficiency of the Jetson Nano during training. Optimizing power usage is crucial for embedded systems to prolong device lifespan and reduce operational costs. Memory utilization metrics offer insights into memory usage patterns, allowing us to potentially track memory-related issues during training.

4.3.2. Metrics Collection on The Desktop Environment

To achieve a similar quality of metrics, for the desktop environment python libraries like "psutil"[24] and "pynvml" (a Python library for collecting Nvidia GPU metrics), alongside the NVIDIA System Management Interface (nvidia-smi)[25] are used to collect metrics regarding CPU and GPU usage. The following metrics were collected and analyzed on the desktop environment: CPU and GPU Utilization, CPU and GPU Temperature, CPU and GPU Power consumption, and overall Memory Utilization.

The collected metrics have the same importance as the ones for the embedded systems. They can serve as a basis for future data-driven decision-making, allowing for more informed optimizations, and opportunities for troubleshooting performance issues.

5. Results and Discussion

5.1. Reward Plots for Classic Q-Learning and Optimized Q-Learning Using Numba

The reward plot for both the Q-Learning and Numba Q-Learning implementations were generated by training the agents for a total of 15000 episodes. These plots provide insights into the agents' learning progress and performance over the course of training.

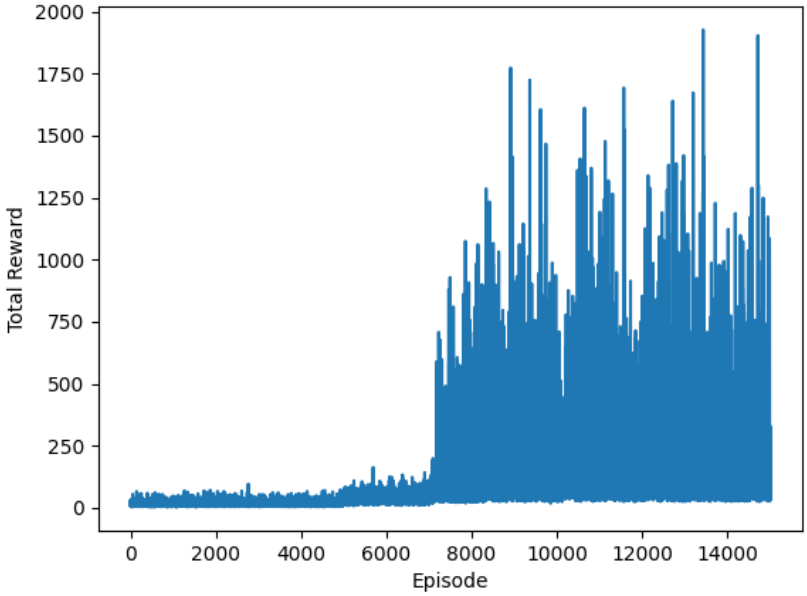

Figure 2. Q-Learning and Numba Q-Learning reward plot (CPU)

Within our implementation, the ϵ value in the ϵ-greedy action selection algorithm begins to decline at the halfway point during training. The reward plot clearly illustrates the distinction between the exploration and exploitation phases of the agents' learning process. During the initial episodes, the agent is in the exploration phase, where it is actively exploring the custom environment and learning optimal strategies. This is characterized by lower reward values, typically ranging between 20 and 200 reward points.

During this exploitation phase, the agent consistently achieve higher reward values compared to the exploration phase. The reward plots show that after the 7000th episode, the agent achieves between 500 and 1900 reward points on average. This indicates that the agent has successfully learned to exploit the environment to maximize rewards using the strategies acquired during the exploration phase.

Using our custom built visualizer to assess the performance of the agent, we observed that the agent successfully balanced the inverted pendulum for approximately 14 seconds. During this time the agent kept the pendulum near the middle point of the track, between

20cm and 30cm of the track length (50cm).

5.2. Reward Plots for Deep Q-Learning and Deep Q-Learning with fixed target values

The reward plot for Deep Q-Learning was generated by training the network for 1000 episodes, the network is trained for 100 epochs. The plot for Deep Q-Learning with fixed target values was generated by training the networks for 100 episodes with the same amount of epochs.

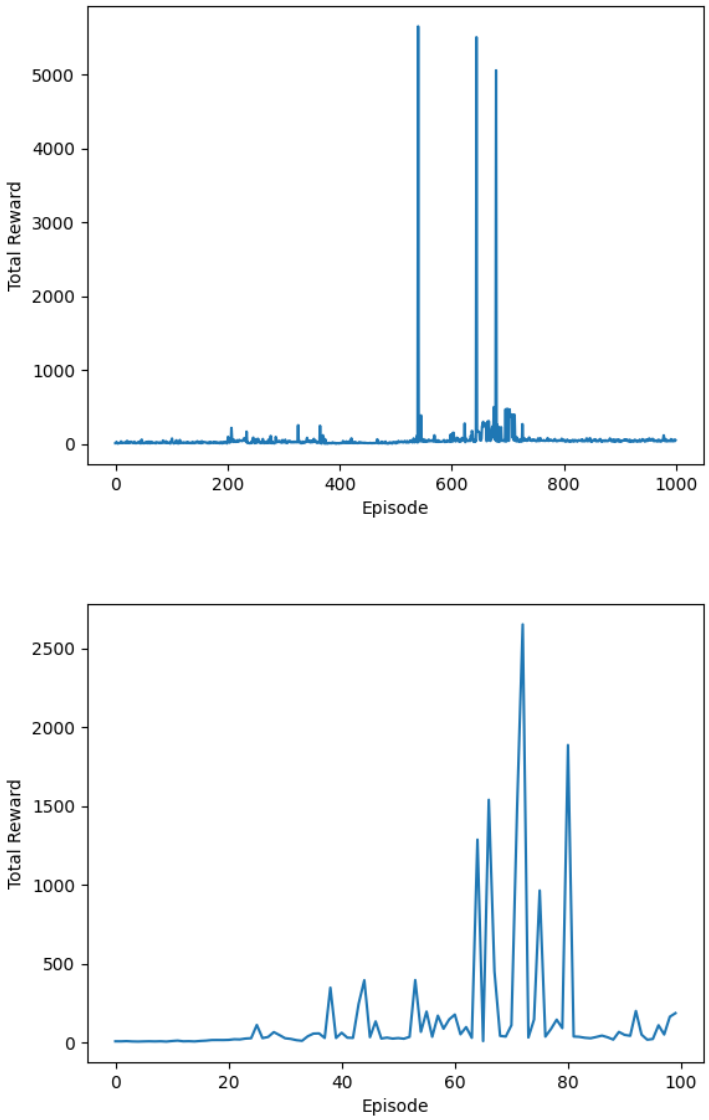

Figure 3. Deep Q-Learning (top) and Deep Q-Learning with fixed target values (bottom) reward plots (GPU)

Within these implementation, the ϵ value in the ϵ-greedy action selection algorithm starts getting reduced at the first quarterly point during training, which can be reflected

in Figure 3.

While with the Q-Learning reward plot we can clearly see a big divide between the exploration and exploitation phases of the learning process, here the difference is less noticeable at first. The agents peak higher when it comes to maximum reward points earned overall, with DQL achieving 5900 points at it's peak while DQL-T achieving 2600 points at it's peak. In comparison, the Q-Learning agent which reached a maximum of 1900 points but had much more consistency in reaching higher rewards during training.

The results of running the agents on the visualizer showcased the differences in the agents learned strategies between implementations. For DQL, the agent managed to keep the inverted pendulum balanced for about only 4 - 5 seconds. It accomplished this by rapidly moving between the near end points of the entire track, between $5cm$ and $45cm$ of the track length ($50cm$). The environment had the ends of the track marked as terminal states, as so the agent completely stopped once it hit one of the edges. The DQL agent seemed to struggle with the task and adopted a much less precise strategy in comparison to the Q-Learning agent. The agent's learned strategy seemed to be to constantly move the cart in the direction that the pendulum was leaning at, which would've been a better strategy if not for the terminal states at the ends of the track length.

For DQL-T, the agent managed to keep the inverted pendulum balanced for around 10 - 11 seconds, a much better result than what the DQL agent achieved. The agent moved the cart much less rapidly than the DQL agent did, slowly moving the cart from the middle of the track ($25cm$ of the track length) to the leftmost side ($5cm$ of the track length) in the first 8 - 9 seconds of balancing. Then the agent proceeded to balance the pendulum for few more seconds before hitting a terminal state and stopping.

5.3. Benchmark Metrics

5.3.1. Jetson Nano

The benchmark process for the embedded measures the CPU, GPU, and memory utilization, along with temperatures of aforementioned processing units and their respective power utilization in watts.

This is done in order to cover one of our main goals for our research: to determine the differences in computational load, energy consumption, and where the discrepancies truly lay. These metrics were measured by simply creating two processes, one for the actual training and another for taking one-shot measurements of device status, as previously stated, in a scheduled manner. The two processes run alongside each other, with the benchmarking process having little to no effect on the actual execution of the algorithms/networks.

The metrics in question aim to display the overall efficacy and effect training has on the respective system, measuring values of computational/memory load in percent, temperature in degrees Celsius, and power consumption using watts.

5.3.2. Average Benchmark Metrics

Table 5. Average Metrics for Different Training Types

Type	CPU (%)	GPU (%)	Mem (%)	CPU Temp (C)	GPU Temp (C)	CPU Pwr (W)	GPU Pwr (W)
CLASSIC	22.59	0.00	0.38	46.34	44.61	1.22	0.0
NUMBA	22.93	0.00	0.41	46.52	44.64	1.22	0.0
DQL_GPU	22.71	7.83	0.61	47.68	44.36	1.39	0.96
DQL_CPU	23.07	0.00	0.42	47.24	45.66	1.26	0.0
DQL_T_GPU	22.53	6.28	0.62	48.06	44.71	1.36	0.98
DQL_T_CPU	23.13	0.00	0.42	46.92	45.46	1.25	0.0

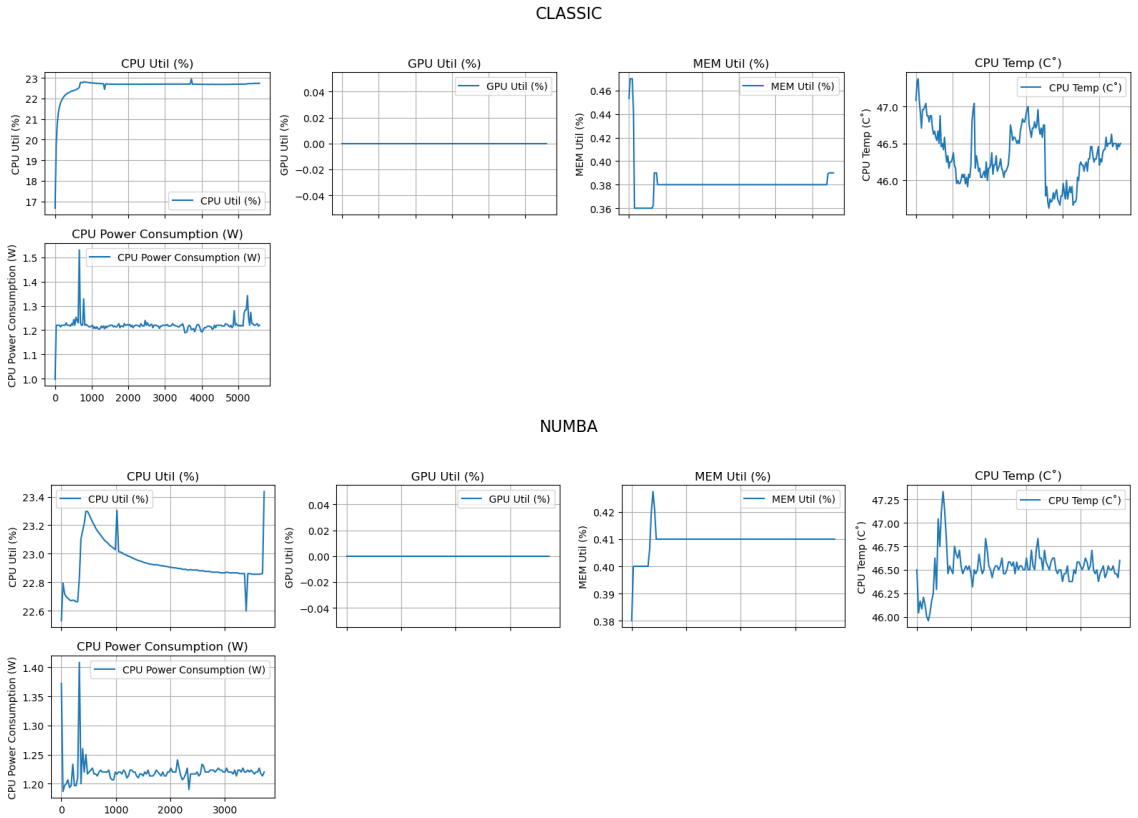

Figure 4. Classic Q-Learning Metrics (top) and Numba Q-Learning Metrics (bottom) reward plots (CPU)

The Figure 4 shows benchmark results for classic Q-Learning and Numba optimized Q-Learning. Both of these algorithms exclusively utilize CPU computation, with the classic version maintaining, on average, 22.6% CPU of the CPU and the Numba version 22.9%, but ever so slightly trending downwards with respect to time. In any other measurements the two algorithms essentially performed almost the same, with the Numba version being slightly more consistent and less sporadic with fluctuations in temperature and memory utilization, but varying more in the power consumption of the CPU. However, the Numba

optimization was approximately 30 minutes faster than the standard, unoptimized version. This proved to be a significant improvement with little effort, yielding an improvement of 33% from the classic Q-Learning to the Numba Q-Learning, and altogether the classic and Numba approach took 1h and 33min and 1h and 2min to complete, respectfully.

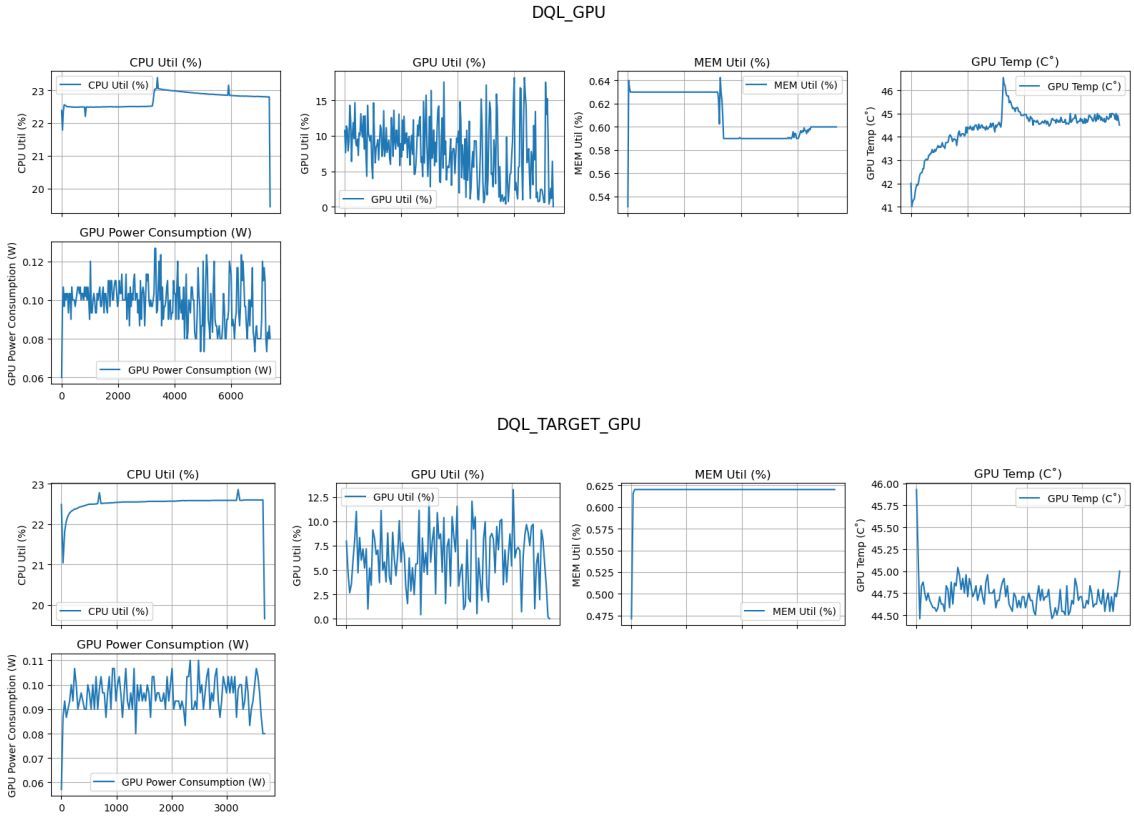

Figure 5. Deep Q-Learning Metrics (top) and Deep Q-Learning with Target Network Metrics (bottom) (GPU)

Our next batch of measurements (Figure 5), for Deep Q-Learning and Deep Q-Learning using a target network, were GPU-centric, as the general purpose of these trials were conducted with parallel computing in mind, based on neural networks and neural networks utilizing a target network. These networks, albeit not particularly complex or large, took a significant amount of time to complete. The processes required to perform neural network training are inherently more complex and typically introduce more overhead, which, for our use-case, did not seem to offer any advantage in either training time or resource utilization. Standard Q-learning proved to be more data-efficient in an autonomous environment such as this, where the state and action spaces are relatively small and well-defined. In contrast, our DQL approach seemed to require a larger amount of interaction with the environment, even though the state-space was relatively small, thereby increasing the training duration. The model training was reduced to 8 epochs for these implementations due to measuring convenience.

On average, the standard DQL approach and DQL with a target network essentially performed the same, with regards to resource utilization. The DQL approach using a

target network seemed to have a stronger regression towards its mean when it comes to temperature and power consumption, indicating a stronger stability in the operation of the system. The DQL using a target network also completed training 50.12% times faster than the regular DQL training, which is a significant improvement in time. We believe this displays a possible improvement in hardware utilization between the two, and potentially less overhead between the CPU and GPU.

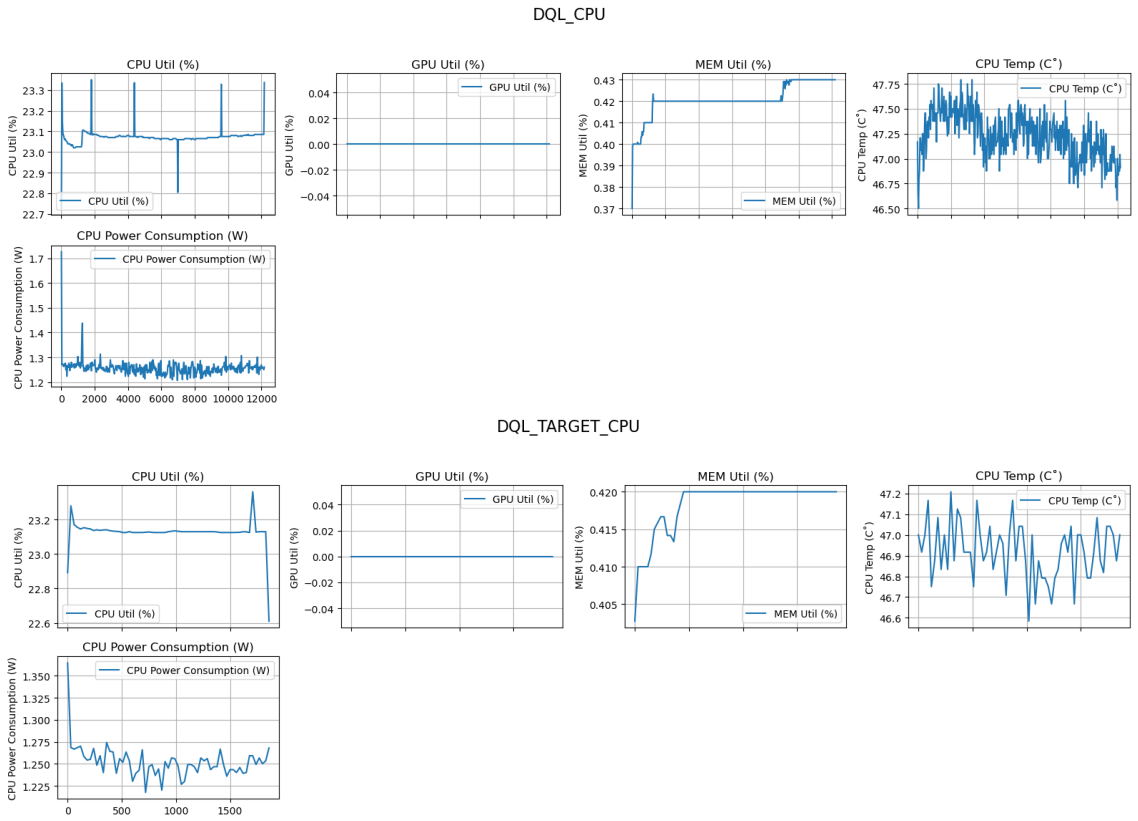

Figure 6. Deep Q-Learning Metrics (top) and Deep Q-Learning with Target Network Metrics (bottom) (CPU)

The final batch of measurements (Figure 6) are for the DQL and DQL using a target network but instead ran exclusively on the CPU. These essentially do the same thing the GPU-centric version does, but on the CPU instead. The standard DQL implementation took the longest to complete out of the six training types, and the DQL using a target network was the *fastest* out of them all. This is interesting, but not all that surprising considering the nature of our autonomous system.

The CPU version of the DQL training used less memory and less power, on average, than the GPU version, with the target network implementation scoring lower on resource usage overall with a better time delta.

5.3.3. Machine Learning Server

The benchmark process for the machine learning server measures the CPU, GPU, and memory utilization, along with temperature for the GPU and measurements of aforementioned processing units and their respective power utilization in watts.

5.3.4. Average Benchmark Metrics

Table 6. Average Metrics for Different Training Types

Type	CPU (%)	GPU (%)	Mem (%)	CPU Temp (C)	GPU Temp (C)	CPU Pwr (W)	GPU Pwr (W)
CLASSIC	100.24	0.00	1.76	64.99	42.59	33.53	0.00
NUMBA	100.32	0.00	1.93	63.91	40.83	34.20	0.00
DQL_GPU	106.47	0.01	4.09	58.88	49.56	37.25	0.01
DQL_CPU	106.44	0.00	2.00	59.64	41.55	36.11	0.00
DQL_T_GPU	106.67	0.02	4.06	57.55	49.43	37.26	0.02
DQL_T_CPU	107.28	0.00	1.96	59.74	40.28	36.30	0.00

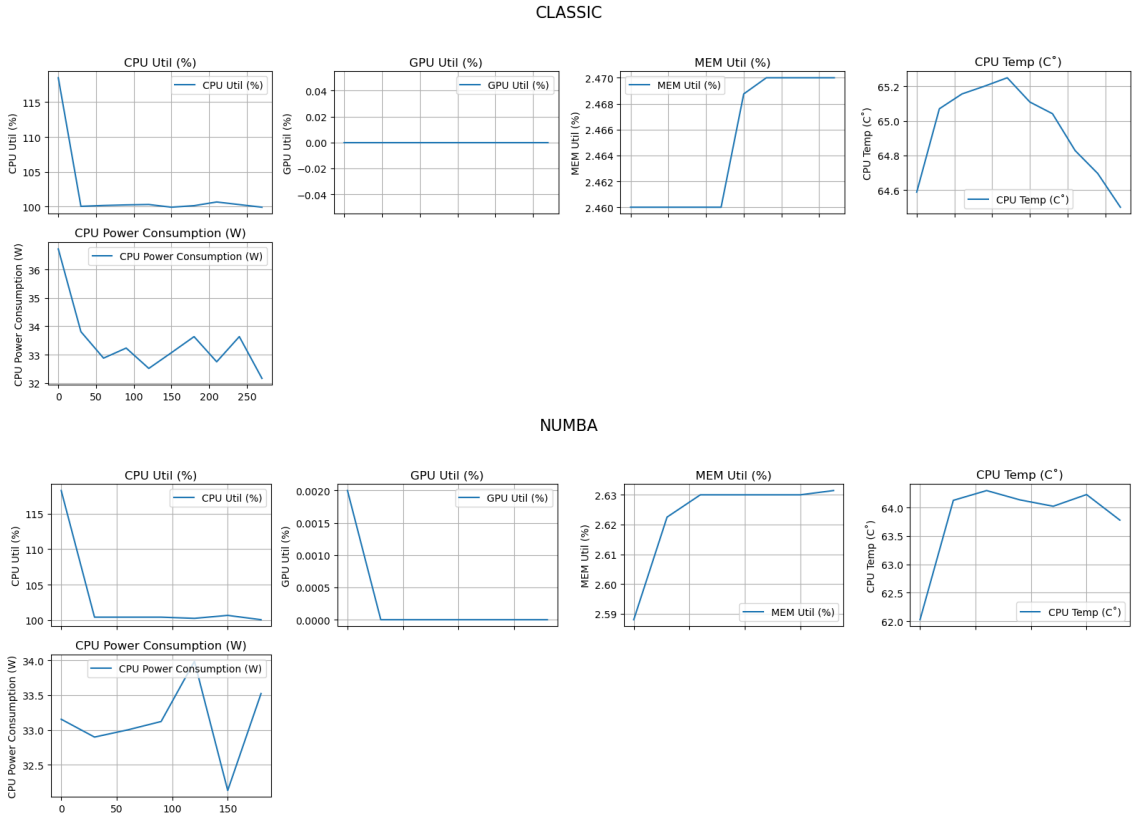

Figure 7. Classic Q-Learning Metrics (top) and Numba Q-Learning Metrics (bottom) reward plots (CPU), ML Server

Same as the previous metrics, the first batch of training was conducted using the Q-Learning implementations. On Figure 7 we can again see that these implementations exclusively utilize CPU computation, albeit with significantly more computational power and speed.

It is clear to see that the machine learning server used an abundant amount of execution power regardless of the training type, and it could be worth questioning if the speed

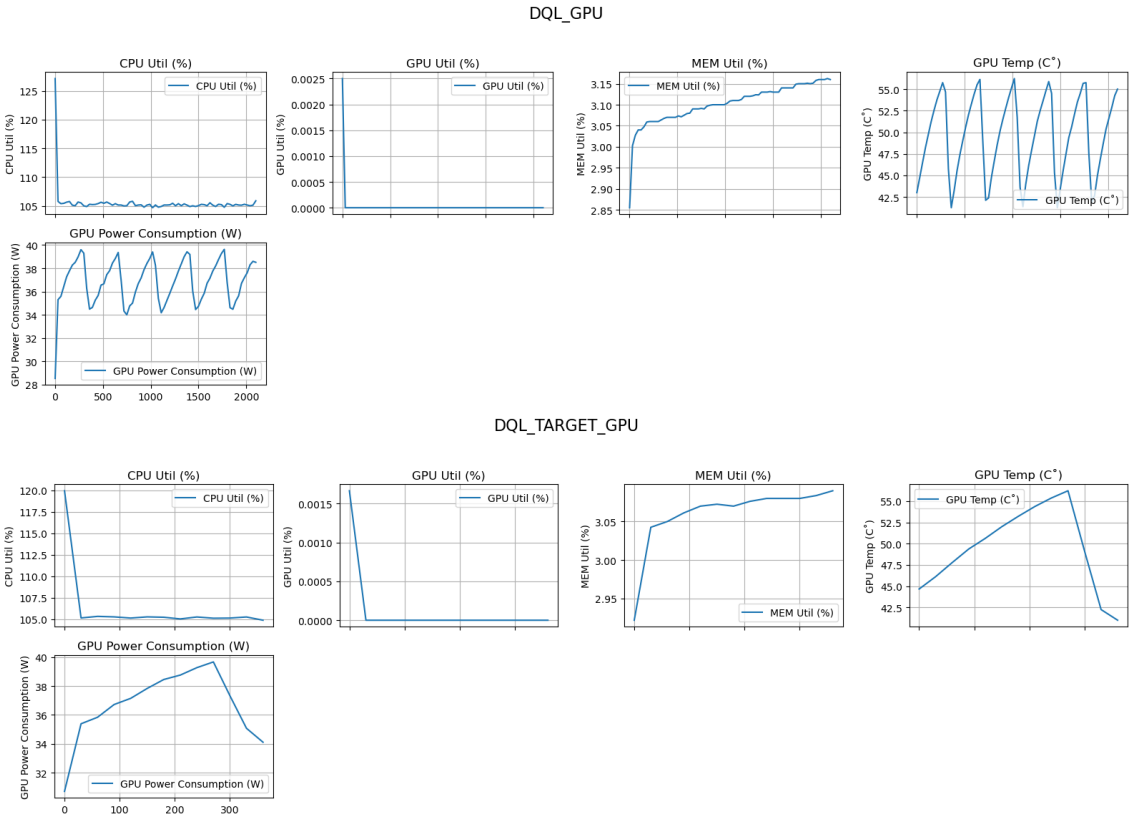

Figure 8. Deep Q-Learning Metrics (top) and Deep Q-Learning with Target Network Metrics (bottom) (GPU), ML Server

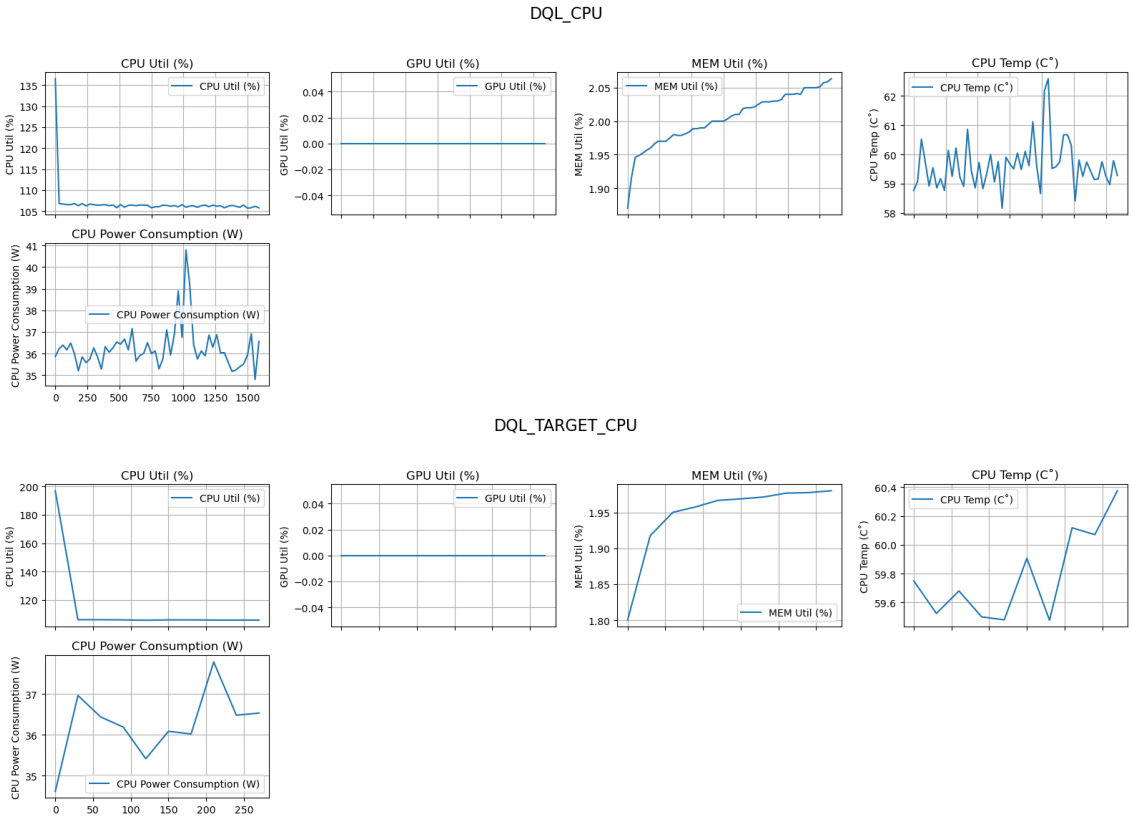

Figure 9. Deep Q-Learning Metrics (top) and Deep Q-Learning with Target Network Metrics (bottom) (CPU), ML Server

at which the training was finished outweighs the environmental and energy impact it has. Nonetheless, these results only become interesting when compared with the energy-to-work ratio of the embedded system.

5.4. Performance per Watt for Embedded System and Machine Learning Server

This metric serves as a general indicator of energy efficiency in most computational systems. It not only helps in assessing the energy consumption relative to the computational output but also helps in making informed decisions about hardware utilization in energy-sensitive environments. In our case, in the context of training various reinforcement learning models, measuring performance per watt allows for a comparative and more objective analysis of energy efficiency across different hardware configurations and algorithms. The formula used to compute this metric involves several steps:

1. The average power consumption (in watts) is calculated as the mean of power consumption readings during the training session
2. The total training duration is converted from seconds to hours.
3. Multiplying the above two yields the total energy consumed during the session in watt-hours
4. Finally, this energy consumption figure is used to determine how many training episodes (or computational tasks) are completed per watt-hour, calculated as $\frac{\text{Total Episodes}}{\text{Total Energy Consumed}}$

This ratio provides a clear and quantitative measure of energy efficiency, highlighting how effectively a system converts electrical power into computational results.

Table 7. Comparison of Performance Per Watt Across Devices

Training Type	**Device**	**CPU Performance Per Watt**	**GPU Performance Per Watt**
Classic	Jetson	7915.26	-
	Server	6100.20	-
Numba	Jetson	11879.74	-
	Server	8534.13	-
DQL GPU	Jetson	352.60	496.49
	Server	47.29	4.61
DQL CPU	Jetson	235.32	-
	Server	62.90	-
DQL Target GPU	Jetson	72.11	102.22
	Server	27.53	2.69
DQL Target CPU	Jetson	154.05	-
	Server	40.32	-

A few interesting observations arose from these results. The performance per watt, which as previously stated essentially measures the executed number of episodes per watt-hour, is clearly different in the two systems in the sense that the Jetson Nano attains far greater results in this metric than its power-house counterpart. Even though execution times are superior on the server, the power utilization of the Jetson Nano far outweighs it. This is obvious in both cases, CPU- and GPU-centric training, considering the embedded system far better utilizes its GPU capabilities.

While a powerful desktop or machine learning server may complete tasks such as this

more quickly due to its more powerful hardware, this speed comes at the cost of efficiency. The Jetson Nano, while slower, uses much less power per unit of work, making it more suitable for applications where energy consumption is a constraint. Potentially combining this knowledge with further optimization techniques could render the embedded systems, or energy-constrained environments or applications where operating costs need to be minimized (edge computing, mobile devices, etc.), a compelling option with the right domain knowledge and techniques for optimization. Conversely, in scenarios where time-to-completion is critical, the server might be the preferred choice despite its lower energy efficiency.

5.5. Execution Times for Embedded System and Machine Learning Server

On Table 8 and Table 9 we can see the execution times for the implementations on the embedded system and the machine learning server.

Training Type	Execution Time (hh:mm:ss)
CLASSIC	01:33:08
NUMBA	01:02:04
DQL_GPU	02:02:50
DQL_CPU	03:23:04
DQL_T_GPU	01:01:16
DQL_T_CPU	00:31:07

Table 8. Execution times for the Jetson Nano.

For the embedded system the difference between the implementation with and without Numba optimization is around 30 minutes, which can be considered significant in the context of embedded systems with constrained resources. By leveraging Numba's JIT compilation, the Q-Learning implementation benefits from low-level optimizations and accelerated code execution, leading to faster training iterations and reduced overall execution time.

For the Deep Q-Learning implementation with fixed value targets on the embedded system, we can see that the implementations performed better while running on the embedded system CPU. Between running the implementation on the GPU and CPU, the overall difference in execution time was also around 30 minutes. This reduction in performance could be caused by several factors. Depending on the memory architecture of the embedded system, the CPU may have better access to memory bandwidth, allowing for faster data retrieval and computation compared to the GPU. The implementation might not utilize the parallel computing aspect of the system GPU to its full potential, the overhead associated with transferring data between the CPU and GPU memory may contribute to the increased execution time when running on the GPU. It's important to note that during training we made sure no other process was using the system's GPU.

This highlights the importance of considering hardware characteristics, memory architecture, optimization techniques, and data transfer overhead in determining the most suitable execution environment for deep reinforcement learning tasks on embedded systems.

Alternatively, for the Deep Q-Learning, we can see that the implementation performed better when running on the system's GPU. Between running the implementation on the GPU and CPU, the overall difference in execution time was also around 1 hour and 20 minutes. Since this implementation involves only one neural network and doesn't require memory swapping between two neural networks like the previous implementation, it could be that this variant of Deep Q-Learning is more suitable for running on GPU-enabled systems. Another point to consider is that the resulting agent from this implementation generally performed worse than the resulting agent from DQL_T, while also having a longer execution time than both DQL_T implementations.

Training Type	Execution Time (hh:mm:ss)
CLASSIC	00:04:24
NUMBA	00:03:05
DQL_GPU	00:34:03
DQL_CPU	00:26:25
DQL_T_GPU	00:05:51
DQL_T_CPU	00:04:06

Table 9. Execution times for the Machine Learning Server.

In comparison to running the implementations on the embedded system, the execution times on the machine learning server are, as expected, drastically smaller. All execution times are improved by 1 hour, with the biggest improvement being the DQL implementation (around 1 hour and 30 minutes). The two deep q-learning algorithms were both primarily executed on the CPU as the complexity of the networks was far too low in order to necessitate the need for utilizing the parallell processing powers of the 4070Ti to its full extent, which could be attributed to the vast amounts of memory and CPU processing powers of the machine. It did, however, at all times use 10254MiB, the max amount, of memory, likely due to the fact that tensorflow automatically allocates all of the available GPU memory to the running CUDA process unless config.gpu_options.allow_growth=True is explicitly set. Doing this, the approximate memory usage on the GPU hovers around 340MiB. Furhtermore, increasing the batch size to 256 units and adding more layers to the neural network does increase GPU utilization to average at 1%, which rules out any potential misconfigurations with Cuda or Tensorflow.

On Table 9, we observe that the difference between classic Q-Learning and Numba Optimized Q-Learning implementations is less significant, suggesting that the optimizations provided by Numba have less impact in a high-performance computing environment compared to an embedded system.

We can see that execution time is a big trade-off when training models directly on

embedded systems. Even so, considering that both systems produced adequate results with agents that solve the problem of balancing an inverted pendulum, the execution times underscore the importance of selecting the appropriate computing platform based on the specific requirements and constraints of the task at hand. While training models directly on embedded systems may imply longer execution times due to hardware limitations, it offers advantages such as real-time inference capabilities, direct access to the agents themselves, and potentially better energy efficiency and performance per watt.

6. Conclusion

The thesis explored the implementation, performance evaluation, and bench-marking of different Q-Learning and Deep Q-Learning variants of reinforcement learning, on both embedded systems and high-performance computing environments. Through empirical experiments, literature research, and analysis, we have examined the trade-offs and implications of training models on different computing platforms, with a focus on solving the problem of balancing an inverted pendulum. The thesis also explores different methods of optimizing performance and resource consumption.

Our findings highlight the diverse challenges and opportunities presented by training models directly on embedded systems versus leveraging high-performance computing setups. The thesis specifically focuses on execution times, agent performances, resource utilization, possible software and algorithmic optimizations, and possible hardware improvements by using GPU-enabled systems for Q-Learning and Deep Q-Learning. Our conclusions on these points are as follows:

- When it comes to execution time trade-offs, training models directly on embedded systems incur longer execution times due to hardware limitations, in comparison to high-end computing setups. However, with appropriate techniques for improving execution time and the domain knowledge of the underlying systems capabilities, the amount of power per unit of work can be greatly reduced and offer a compelling alternative to training outside of the target device or transfer learning. The best-performing implementations on the embedded system were the classic Q-Learning implementation with Numba JIT compilation and the Deep Q-Learning variant with fixed target values (running on the system CPU).

- Observing the execution time metrics, the promising performance of aforementioned implementations suggests that with software optimizations, and choosing Deep Q-Learning variants that improve the training times of agents, embedded systems can still be viable platforms for training and deploying reinforcement learning implementations. Regarding regular Deep Q-Learning, execution time on the embedded system improved by running the implementation on the system GPU. This highlights the potential benefits of leveraging specialized hardware, such as GPUs, to accelerate training and improve the efficiency of reinforcement learning algorithms in resource-constrained environments.

- Despite longer execution times, models trained on embedded systems can still produce good-performing agents that successfully solve complex tasks, such as the inverted pendulum problem.

- While high-performance computing setups offer faster training times it comes at the expense of increased resource requirements. The differences in computational load and energy consumption highlight the better efficiency of resource utilization by the embedded system during training. However, high-performance computing setups are better for pre-training models that can then be deployed on embedded systems to perform actual work, due to their faster training times and higher resources.

- The choice of computing platform influences the effectiveness of hardware and software optimizations. Embedded systems benefit more from software optimizations and improved Deep Q-Learning variants. Comparing the execution times of Deep Q-Learning implementations on the system's GPU and CPU, we observed that GPU execution resulted in improved performance for Deep Q-Learning, and decreased performance for the variant with fixed target values. This could be attributed to factors such as memory bandwidth limitations or sub-optimal utilization of GPU resources, among other reasons. This can perhaps be further explored in future work.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI].

[2] R. Jurdak, A. Elfes, B. Kusy, *et al.*, "Autonomous surveillance for biosecurity," *Trends in biotechnology*, vol. 33, no. 4, pp. 201–207, 2015.

[3] R. Eaton, J. Katupitiya, K. W. Siew, and B. Howarth, "Autonomous farming: Modelling and control of agricultural machinery in a unified framework," *International Journal of Intelligent Systems Technologies and Applications*, vol. 8, no. 1-4, pp. 444–457, 2010. DOI: 10.1504/IJISTA.2010.030223. eprint: https://www.inderscienceonline.com/doi/pdf/10.1504/IJISTA.2010.030223. [Online]. Available: https://www.inderscienceonline.com/doi/abs/10.1504/IJISTA.2010.030223.

[4] Y. Gu, J. C. Goez, M. Guajardo, and S. W. Wallace, "Autonomous vessels: State of the art and potential opportunities in logistics," *International Transactions in Operational Research*, vol. 28, no. 4, pp. 1706–1739, 2021.

[5] M. Tavakoli, J. Carriere, and A. Torabi, "Robotics, smart wearable technologies, and autonomous intelligent systems for healthcare during the covid-19 pandemic: An analysis of the state of the art and future vision," *Advanced Intelligent Systems*, vol. 2, no. 7, p. 2 000 071, 2020. DOI: https://doi.org/10.1002/aisy.202000071. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/aisy.202000071. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202000071.

[6] M. Hillebrand, M. Lakhani, and R. Dumitrescu, "A design methodology for deep reinforcement learning in autonomous systems," *Procedia Manufacturing*, vol. 52, pp. 266–271, 2020, System-Integrated Intelligence – Intelligent, Flexible and Connected Systems in Products and ProductionProceedings of the 5th International Conference on System-Integrated Intelligence (SysInt 2020), Bremen, Germany, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2020.11.044. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978920321879.

[7] T. Zhang and H. Mo, "Reinforcement learning for robot research: A comprehensive review and open issues," *International Journal of Advanced Robotic Systems*, vol. 18, no. 3, p. 17 298 814 211 007 305, 2021. DOI: 10.1177/17298814211007305. eprint: https://doi.org/10.1177/17298814211007305. [Online]. Available: https://doi.org/10.1177/17298814211007305.

[8] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698. [Online]. Available: https://doi.org/10.1007/BF00992698.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[10] S. Spanò, G. C. Cardarilli, L. Di Nunzio, *et al.*, "An efficient hardware implementation of reinforcement learning: The q-learning algorithm," *IEEE Access*, vol. 7, pp. 186 340–186 351, 2019. DOI: 10.1109/ACCESS.2019.2961174.

[11] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, fpgas, and end-devices: A survey," *Electronics*, vol. 8, no. 11, 2019, ISSN: 2079-9292. DOI: 10.3390/electronics8111289. [Online]. Available: https://www.mdpi.com/2079-9292/8/11/1289.

[12] F. Svoboda, D. Nunes, M. Alizadeh, *et al.*, "Resource efficient deep reinforcement learning for acutely constrained tiny ml devices," in *Research Symposium on Tiny Machine Learning*, 2021. [Online]. Available: https://openreview.net/forum?id=_vo8DFo9iuB.

[13] T. Szydlo, P. P. Jayaraman, Y. Li, G. Morgan, and R. Ranjan, "Tinyrl: Towards reinforcement learning on tiny embedded devices," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, ser. CIKM '22, Atlanta, GA, USA: Association for Computing Machinery, 2022, pp. 4985–4988, ISBN: 9781450392365. DOI: 10.1145/3511808.3557206. [Online]. Available: https://doi.org/10.1145/3511808.3557206.

[14] S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, ser. AIChallengeIoT '20, Virtual Event, Japan: Association for Computing Machinery, 2020, pp. 7–13, ISBN: 9781450381345. DOI: 10.1145/3417313.3429378. [Online]. Available: https://doi.org/10.1145/3417313.3429378.

[15] S. Kato, S. Tokunaga, Y. Maruyama, *et al.*, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 287–296. DOI: 10.1109/ICCPS.2018.00035.

[16] M. Saravanan, P. S. Kumar, and A. Sharma, "Iot enabled indoor autonomous mobile robot using cnn and q-learning," in *2019 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*, 2019, pp. 7–13. DOI: 10.1109/ICIAICT.2019.8784847.

[17] A. Sharma, K. Xu, N. Sardana, *et al.*, *Autonomous reinforcement learning: Formalism and benchmarking*, 2022. arXiv: 2112.09605 [cs.LG].

[18] I. Jang, H. Kim, D. Lee, Y.-S. Son, and S. Kim, "Knowledge transfer for on-device deep reinforcement learning in resource constrained edge computing systems," *IEEE Access*, vol. 8, pp. 146 588–146 597, 2020. DOI: 10.1109/ACCESS.2020.3014922.

[19] A. Veletanlic and D. Madzovic, *A simulation of an inverted pendulum robot, made in python*, [Accessed 18-03-2024], 2024. [Online]. Available: https://github.com/HKR-Thesis/inverted_pendulum_simulator.

[20] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[21] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022.

[22] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[23] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[24] G. Rodola, *Psutil documentation*, 2020.

[25] D. Hope, "Measuring gpu power consumption using nvidia tools," 2022.

Appendices

Appendix A. Custom Visualizations

Bellow are links for videos showcasing our visualization of the agents.

Q-Learning agent, trained for 15000 episodes:

`https://youtube.com/shorts/k2z1yEMpFkc`

Deep Q-Learning agent, trained for 1000 episodes with 100 epochs during training:

`https://youtube.com/shorts/Bib0GPwKCww`

Deep Q-Learning with fixed Q-Value targets agent, trained for 100 episodes with 100 epochs during training:

`https://youtube.com/shorts/CsGVDGd6Jdk`