



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Departamento de Engenharia Informática



Learning On-device for Autonomous Game Playing

GONÇALO ALEXANDRE DE MATOS FERNANDES
(Licenciado)

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientador:

Doutor Mário Pereira Véstias

Júri:

Presidente: Doutor Tiago Miguel Braga da Silva Dias

Vogais:

Doutor Rui Manuel Feliciano de Jesus

Doutor Mário Pereira Véstias

December, 2025

Learning on-device for Autonomous Game Playing

GONÇALO ALEXANDRE DE MATOS FERNANDES
(Licenciado)

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador:

Doutor Mário Pereira Véstias, IPL/ISEL

Júri:

Presidente: Doutor Tiago Miguel Braga da Silva Dias, IPL/ISEL

Vogais:

Doutor Rui Manuel Feliciano de Jesus, IPL/ISEL

Doutor Mário Pereira Véstias, IPL/ISEL

December, 2025

Acknowledgments

I would like to express my sincere gratitude to all those who supported me throughout the development of this thesis and helped me conclude this final chapter of my academic journey.

First, I thank the Instituto Superior de Engenharia de Lisboa (ISEL) for providing the academic environment, resources, and opportunities that made this work possible.

I am especially grateful to my adviser, Professor Mário Véstias, for his guidance, insightful feedback, and continuous support. His expertise and encouragement were fundamental in shaping this project and ensuring its successful completion.

I also wish to thank my colleagues António Goulão, José Jorge, and Pedro Apolinário, whose discussions, suggestions, and exchange of ideas played an important role in the progress of this work.

Finally, special thanks to my friends and family for their patience, encouragement, and support over the course of this thesis. Their understanding and motivation allowed me to persevere through the challenges faced along the way.

To all who contributed, directly or indirectly, I extend my heartfelt appreciation.

Statement of integrity

I declare that this project work is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Lisbon, 26 December 2025

Resumo

A presente tese investiga a execução de algoritmos de Deep Reinforcement Learning (DRL) em sistemas embebidos para jogar videogames de forma autónoma. No decorrer dos anos, os algoritmos de DRL demonstraram desempenhos elevados numa variedade de tarefas complexas, mas as suas exigências computacionais limitam a sua aplicabilidade em sistemas com recursos reduzidos. Este trabalho explora a viabilidade de adaptar e otimizar métodos de DRL para execução em hardware embebido.

Foram selecionados, implementados e avaliados num ambiente de trabalho uma vasta seleção de algoritmos, pertencentes às famílias policy-based, value-based e actor-critic, antes de serem transferidos para a plataforma embebida NVIDIA Jetson Orin Nano. Foram também aplicadas múltiplas estratégias de otimização, incluindo ajustes na representação dos dados, afinamento de hiperparâmetros, operações de precisão mista e gestão de memória. A avaliação destes algoritmos decorreu sobre três ambientes de jogo de Atari com uma complexidade crescente, o Pong, o Breakout e o Space Invaders.

Os resultados demonstraram que os algoritmos de DRL executados em dispositivos embebidos podem alcançar desempenhos acima da média humana (p.ex., 74 pontos no Breakout utilizando uma DDQN, face à média de 31 pontos) em tempos de treino inferiores a 8 horas. Entre os métodos testados, o Double Deep Q-Network (DDQN) obteve os melhores resultados, o Proximal Policy Optimisation (PPO) revelou-se o mais leve em termos de consumo de recursos (nunca ultrapassando os 2 GB de Random Access Memory (RAM)) e o Deep Q-Network (DQN) apresentou um compromisso equilibrado entre precisão e eficiência, com pontuações até 37 pontos e um consumo de memória de somente 3 GB.

Este trabalho estabeleceu uma framework reutilizável para treinar e implementar algoritmos de DRL em dispositivos embebidos, constituindo uma base para futuros avanços na aprendizagem por reforço e nas suas aplicações para sistemas autónomos.

Palavras-chave

Sistemas Embebidos; Deep Reinforcement Learning; Otimização de Algoritmos, Ambientes de Jogo, Aprendizagem no Dispositivo

Abstract

This thesis investigates the deployment of Deep Reinforcement Learning (DRL) algorithms on embedded devices for autonomous game playing. Over the years, DRL has demonstrated remarkable performance in a variety of complex tasks, but its computational demands often restrict its applicability to resource-constrained systems. This work explores the feasibility of adapting and optimising DRL methods for execution on embedded hardware.

A diverse set of algorithms, including policy-based, value-based, and actor–critic families, were selected, implemented, and benchmarked in a desktop environment before being deployed on a NVIDIA Jetson Orin Nano embedded platform. Multiple optimisation strategies were applied, including data representation adjustments, hyperparameter tuning, mixed precision operations, and memory management. The evaluation was conducted across three Atari game environments of increasing complexity, Pong, Breakout, and Space Invaders.

The results demonstrated that DRL algorithms executed on embedded devices can achieve performances above human averages on some games (e.g., a score of 74 on Breakout using a DDQN, compared to the human average of 31), all within reasonable training times of below 8 hours. Among the tested methods, Double Deep Q-Network (DDQN) obtained the strongest overall results, Proximal Policy Optimisation (PPO) proved the most resource-efficient method (never surpassing 2 GB of RAM), and Deep Q-Network (DQN) offered a balanced compromise between accuracy and efficiency, with scores going up to 37 and a memory consumption of only 3 GB.

This work established a reusable framework for training and deploying DRL on embedded devices, providing a foundation for further advancements in reinforcement learning and its applications in real-world autonomous systems.

Keywords

Embedded Devices; Deep Reinforcement Learning; Algorithm Optimisation; Game Environments; On-device Learning

Table of Contents

Acknowledgments	iii
Resumo	vii
Abstract	ix
List of Figures	xv
List of Tables	xvii
List of Listings	xix
Acronyms	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Main Contributions	3
1.4 Document Organisation	3
2 Background and Related Work	5
2.1 Neural Networks	6
2.1.1 Fundamentals of a Neural Network	6
2.1.2 Learning Mechanisms in Neural Networks	9
2.2 Deep Learning	11
2.2.1 Foundations of Deep Learning	11
2.2.2 Common Deep Learning Architectures	11
2.3 Reinforcement Learning	13
2.3.1 Fundamentals of Reinforcement Learning	13
2.3.2 Deep Reinforcement Learning	15
2.3.3 Deep Reinforcement Learning Enhancements	16
2.3.4 Applications of Reinforcement Learning	18
2.4 Deep Learning in Embedded Systems	18
2.4.1 Challenges of Deploying Deep Learning on Embedded Systems	19
2.4.2 Techniques for Deploying Deep Learning in Embedded Systems	19

2.4.3	Applications of Deep Learning in Embedded Systems	20
2.5	Algorithm Optimisation	21
2.5.1	Optimisation Techniques for Deep Learning Algorithms	21
2.6	Related Work	23
2.6.1	Reinforcement Learning Algorithms and Performance Comparison	23
2.6.2	Reinforcement Learning in Gaming	24
2.6.3	On-device Reinforcement Learning	25
2.7	Chapter Summary	26
3	Analysis and Evaluation of Deep Reinforcement Learning Algorithms for Embedded Game Playing	27
3.1	Design Space Exploration Framework	28
3.1.1	Overview of the Proposed Methodology	28
3.1.2	Project Requirements and Target Platform	29
3.1.3	Design Framework	30
3.2	Exploring Game Environments	31
3.2.1	Environment Selection	31
3.2.2	Environment Setup and Customisation	33
3.3	Design and Architecture for Training DRL Algorithms	35
3.3.1	Neural Network Architectures	35
3.3.2	Training Platforms	36
3.3.3	Utility Functions	37
3.3.4	Software Architecture	38
3.4	Reinforcement Learning Algorithms	39
3.4.1	Comparison of DRL Algorithms for Game Playing	52
3.5	Optimising DRL Algorithms for Embedded Computing	53
3.6	Chapter Summary	54
4	Evaluation on the Embedded Device	57
4.1	Evaluation Methodology	57
4.2	Results on Selected Environments	58
4.2.1	Low Complexity Environment - Pong	58
4.2.2	Medium Complexity Environment - Breakout	62
4.2.3	High Complexity Environment - Space Invaders	64
4.2.4	Result Analysis and Discussion	66
4.3	Results on Modified Environments	66
4.3.1	Alternative Game Modes for Breakout	67
4.3.2	Cross-Environment Knowledge Transfer	68
4.4	Chapter Summary	68
5	Conclusions and Future Work	71
5.1	Research Recap	71

5.2 Achievements	71
5.3 Limitations	72
5.4 Future Work	72
Bibliography	78

List of Figures

2.1	Structure of a Neural Network Architecture [43]	7
2.2	Reinforcement Learning Scheme	14
2.3	Deep Reinforcement Learning Algorithm Taxonomy	15
3.1	Proposed Methodology Flowchart	28
3.2	NVIDIA Jetson Orin Nano development board [44]	30
3.3	Pong Environment Preview [19]	32
3.4	Breakout Environment Preview [18]	32
3.5	Space Invaders Environment Preview [20]	33
3.6	Software Architecture Diagram	39
3.7	Random Agent Learning Curve	40
3.8	REINFORCE Learning Curve	41
3.9	TRPO Learning Curve	43
3.10	PPO Learning Curve	44
3.11	A2C Learning Curve	45
3.12	SAC Learning Curve	46
3.13	DQN Learning Curve	48
3.14	DDQN Learning Curve	49
3.15	DRQv2 Learning Curve	50
3.16	RDQN Learning Curve	52
4.1	Pong Learning Curve with PPO	59
4.2	Pong Learning Curve with DQN	60
4.3	Pong Learning Curve with DDQN	61
4.4	Breakout Learning Curve with PPO	62
4.5	Breakout Learning Curve with DQN	63
4.6	Breakout Learning Curve with DDQN	63
4.7	Space Invaders Learning Curve with PPO	64
4.8	Space Invaders Learning Curve with DQN	65
4.9	Space Invaders Learning Curve with DDQN	65

List of Tables

3.1	Tested Algorithms Comparison	52
4.1	Embedded Device Algorithm Comparison on the Pong Environment	61
4.2	Embedded Device Algorithm Comparison on the Breakout Environment	64
4.3	Embedded Device Algorithm Comparison on the Space Invaders Environment	66

List of Listings

3.1	Simple CUDA Device Setup	37
3.2	Correct CUDA Device Setup Output	37
3.3	Random Agent Scores	40
3.4	Average REINFORCE Scores	41
3.5	Average TRPO Scores	43
3.6	Average PPO Scores	44
3.7	Average A2C Scores	45
3.8	Average SAC Scores	46
3.9	Average DQN Scores	48
3.10	Average DDQN Scores	49
3.11	Average DRQv2 Scores	50
3.12	Average RDQN Scores	52
4.1	Pong Average Scores with PPO	59
4.2	Pong Average Scores with DQN	60
4.3	Pong Average Scores with DDQN	61
4.4	Breakout Average Scores with PPO	62
4.5	Breakout Average Scores with DQN	63
4.6	Breakout Average Scores with DDQN	63
4.7	Space Invaders Average Scores with PPO	64
4.8	Space Invaders Average Scores with DQN	65
4.9	Space Invaders Average Scores with DDQN	65
4.10	Breakout <i>Catch</i> Average Scores	67
4.11	Breakout <i>Breakthrough</i> Average Scores	68

Acronyms

A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
ACER	Efficient Actor-Critic with Experience Replay
Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDQN	Double Deep Q-Network
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DRQv2	Data-Regularized Q-learning
DSE	Design Space Exploration
FNN	Feedforward Neural Network
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
MBVE	Model-Based Value Expansion
MCTS	Monte Carlo Tree Search
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Networks
NPC	Non Playable Character
PER	Prioritized Experience Replay
PPO	Proximal Policy Optimisation

RAM Random Access Memory
RDQN Rainbow Deep Q-Network
ReLU Rectified Linear Unit
RGB Red-Green-Blue
RL Reinforcement Learning
RNN Recurrent Neural Network
RTS Real-time Strategy
SAC Soft Actor-Critic
SARSA State-Action-Reward-State-Action
SGD Stochastic Gradient Descent
SSP Stochastic Shortest Path
TD Temporal-difference
TRPO Trust Region Policy Optimization
USB Universal Serial Bus
VR Virtual Reality
VRAM Video Random Access Memory

Chapter 1

Introduction

Advancements in Artificial Intelligence (AI), particularly in the subfield of Machine Learning (ML), have led to the development of increasingly autonomous systems in domains such as robotics, natural language processing, and gaming capable of learning and adapting in real time. Reinforcement Learning (RL) is a subset of ML with the ability to enable intelligent agents to learn optimal strategies by interacting with their environment through trial and error. Deep Reinforcement Learning (DRL) combines RL with Deep Neural Networks to further enhance these learning capabilities, allowing networks to process more sophisticated tasks.

Enabling on-device learning with Reinforcement Learning for autonomous game-playing presents an opportunity to develop intelligent systems that can adapt and improve their performance over time without relying on external processing resources. However, deploying these models in embedded systems remains a challenge due to constraints in computational power, lack of memory, and low energy efficiency.

This thesis explores the possibility of optimising and deploying DRL algorithms on embedded hardware systems, to enable high-performance on-device learning without compromising model accuracy. This study contributes to the advancement of autonomous capabilities in embedded systems and the overall performance of AI agents in gaming.

1.1 Motivation

The main motivation for this work arises from the rapid evolution of AI and intelligent autonomous systems, which are increasingly capable of mastering complex tasks, including playing entire games by learning and adapting in real-time. Advances in DRL have led to remarkable breakthroughs in gaming agents, with models surpassing human performance in various environments. However, these advances have largely depended on the use of powerful machines or large-scale infrastructures, limiting their applicability for real-world resource-constrained scenarios.

Bringing DRL capabilities directly into embedded devices is a necessary step to fully unlock the potential of intelligent autonomous systems. On-device learning enables real-time decision-making and reduces the dependency on external infrastructure, but comes with signif-

icant challenges due to their limited computational power and resources. Efficient deployment requires optimising model architectures, reducing computational overhead, and striking a balance between performance, resource efficiency, and accuracy.

This work is motivated by the need to explore these challenges, investigate how to adapt DRL methods to constrained hardware, and demonstrate that embedded systems can achieve meaningful performance without compromising efficiency. Overcoming these limitations will help expand the applicability of RL to a wider range of devices and applications where adaptive intelligence is essential.

1.2 Objectives

The primary objective of this thesis is to design, optimise, and evaluate an embedded system capable of executing DRL algorithms for autonomous game-playing. This system will be designed to learn and adapt in real time while operating within the constraints of embedded hardware. To achieve this goal, the research was structured into the following five objectives:

- **Explore** different gaming environments to identify the most suitable for experimentation, balancing diversity, complexity, and feasibility.
- **Profile** different DRL algorithms by exploring, implementing, and preliminary testing them, understanding their trade-offs in terms of performance and resource requirements and identifying the most suitable approaches for embedded deployment.
- **Optimise** the most promising algorithms to better suit the constraints of embedded devices, improving their computational efficiency without compromising performance.
- **Deploy** the optimised models on the embedded device, ensuring they run effectively within its hardware constraints.
- **Evaluate** each algorithm's performance on the embedded platform, comparing the obtained results, and analysing the trade-offs in accuracy, execution times, and resource usage.

Together, these objectives provide a structured workflow to guide the development of this work, ensuring a systematic approach to achieve the final goal of deploying DRL algorithms to train on resource-constrained devices.

1.3 Main Contributions

The results and findings reported in this thesis have been submitted for peer review as a research article to a prominent venue in the field of DRL for Embedded Computing Systems and System-Level Optimisation.

The source code for this project is available on the GitHub repository¹. The repository contains the agent implementations, model definitions, training and evaluation scripts, and other utilities necessary to reproduce the results reported throughout this thesis.

In addition to the implementations used in the final experiments, the repository also contains several experimental variants and alternative implementations that were explored during the research but not included in the final evaluation.

The explored environments can be found on the Gymnasium website, listed under the "Environments" tab².

1.4 Document Organisation

This document is structured to provide a comprehensive understanding of the research problem, the background knowledge required to understand it, the proposed solution, and the experimental results obtained so far, more specifically:

- **Chapter 1** (Introduction) Establishes the context, motivation, and objectives of the research, along with an overview of the thesis structure.
- **Chapter 2** (Background and Related Work) Reviews the theoretical foundations explored in this thesis, such as neural networks, deep learning, reinforcement learning, the process of deploying and optimising these techniques for embedded systems and prior related works on domains such as reinforcement learning in gaming and on-device learning.
- **Chapter 3** (Analysis and Evaluation of Deep Reinforcement Learning Algorithms for Embedded Game Playing) Details the design choices, algorithm selection, experimental setup, system architecture and the optimisations considered for embedded deployment.
- **Chapter 4** (Evaluation on the Embedded Device) Presents the obtained results for the selected algorithms, including learning curves, performance metrics and resource utilisation analysis on the target embedded platform.
- **Chapter 5** (Conclusions and Future Work) Summarises the findings, discusses limitations, and outlines potential directions for future research and improvements.

¹<https://github.com/gu113/Thesis-Project>

²<https://gymnasium.farama.org/>

Chapter 2

Background and Related Work

The purpose of this chapter is to provide the foundational knowledge and context necessary to understand the main topics presented in this thesis. It begins by exploring all the core technologies, methodologies, and algorithms that were researched to better understand the main focus of this thesis.

In addition, this chapter also reviews some existing research in this field, highlighting most contributions and revealing some knowledge gaps or limitations that this thesis seeks to address. The structure of this chapter is as follows.

- **Section 2.1** provides an in-depth introduction to the fundamental concepts of neural networks, describing their underlying structure, key components, and various learning mechanisms.
- **Section 2.2** focuses on deep learning, offering an overview of its evolution, fundamental concepts, core principles, and prominent architectures.
- **Section 2.3** introduces the key principles of reinforcement learning, its fundamental components, training models, and some common applications of this mechanism.
- **Section 2.4** explores the integration of deep learning into embedded systems, discussing the challenges associated with the integration, and the techniques used to optimise and deploy deep learning models on environments with limited resources.
- **Section 2.5** analyses the trade-offs associated with the implementation of optimisation approaches, followed by an exploration of algorithm optimisation techniques designed to improve model efficiency and reduce computational costs.
- **Section 2.6** presents a review of related work, examining existing research, and advances, analysing previous studies, methodologies, and key contributions in the field, and highlighting both the progress made and the challenges that remain.
- **Section 2.7** concludes the chapter, summarizing the theoretical foundations, contextualizing the research gap identified through the related work, and providing an introduction to the subsequent chapter.

2.1 Neural Networks

Neural networks are computational models designed to emulate the biological neural processes of living beings, using the structure of the human brain and its functionalities as inspiration. These models consist of interconnected layers of artificial neurons (or nodes) that process, analyse, and transform received data, enabling pattern recognition and eventually decision-making capabilities [49].

Neural networks are the foundation of many machine learning techniques and AI systems, allowing machines to perform tasks that range from identifying objects in images to understanding, processing, and generating human language [33].

This section provides an overview of the fundamental principles underlying neural networks, their key components, commonly used architectures, and some examples of practical applications.

2.1.1 Fundamentals of a Neural Network

The overall architecture of a neural network consists of multiple components and processes working together to model complex relationships in data. These components include interconnected layers, weights, and biases, activation functions that introduce non-linearity, and training processes that improve the network's predictive capabilities.

Layers are the foundation of neural networks [33], consisting of multiple interconnected computational units called neurons that work together to detect patterns and features in data, to transform information from raw input to meaningful outputs.

A neural network is organised in a hierarchical structure of layers identified according to their position in the neural network [21], namely:

- **Input Layer:** Receives the input data, serving as an entry point for the network;
- **Hidden Layers:** Process the information received from the input layer through weighted connections, performing transformations on the data, and allowing the network to learn complex patterns and representations. The depth (amount of hidden layers) can vary based on the complexity of the network;
- **Output Layer:** Produces the final results of the network, varying its structure depending on the context of the problem.

These layers function collaboratively to define the structure of a neural network. The input layer serves as the initial interface, receiving raw data and passing it to the hidden layers. The hidden layers will then perform successive transformations to extract the relevant features from the data. Subsequently, these features are passed to the output layer, which interprets the information and generates the network predictions.

The integration of all layers and the respective connections between them defines the architecture of a neural network (see generic structure in Figure 2.1), establishing the foundational structure on which the network learns, adapts, and performs its tasks.

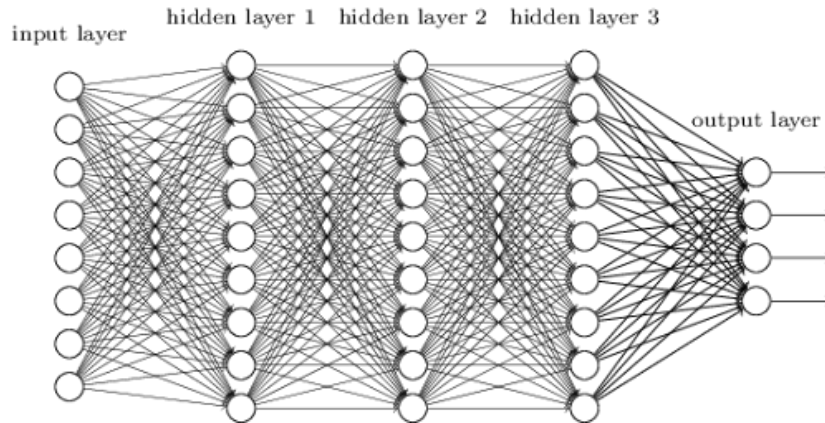


Figure 2.1 Structure of a Neural Network Architecture [43]

As illustrated in the figure, each layer is built upon the concept of connecting artificial neurons, to process and model complex relationships in data. The example in the figure includes three hidden layers, each with multiple neurons.

In neural networks, each neuron within a layer is connected to some or all other neurons in the subsequent layers. These connections facilitate the transmission of information between nodes. To alter, adapt, and refine this information as it propagates through the network, weights and biases are applied to the inputs to generate the outputs of neurons (Y) (as seen in equation 2.1).

$$Y = \sum (weights * inputs) + bias \quad (2.1)$$

The performance and overall effectiveness of neural networks rely on these two fundamental learnable parameters [46].

Weights control the strength of the connections between layers and determine the relative importance of input signals as they propagate through the network. As adjustable parameters, they decide the extent to which input signals influence the network output, allowing the model to learn and adapt during the training process.

Biases act as additional parameters in the network, guaranteeing that neurons remain active even when all input values are zero. By enabling the model to shift the output of the activation functions independently of the input signals, biases provide an additional degree of freedom to the network and enhance its flexibility.

Together, these parameters enable neural networks to learn advanced patterns and relationships within the data by minimising the error between predicted and actual output during the training process.

Activation functions are non-linear transformations that are applied to node outputs, enabling the network to model complex patterns and relationships that are not possible by using only linear or binary step transformations [21]. Some commonly used activation functions include:

- **Sigmoid/Logistic:** Produces output ranging between 0 and 1, typically used in binary

classification tasks [6]. This function is defined mathematically as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

- **ReLU** (Rectified Linear Unit): Provides a nonlinear transformation where values less than zero are set to zero, and positive values remain unchanged, becoming the default activation function in many deep networks [41]. This function is defined mathematically as:

$$f(x) = \max(0, x) \quad (2.3)$$

- **Tanh**: Output values ranging from -1 to 1, useful for data with both positive and negative correlations [6]. This function is defined mathematically as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.4)$$

- **Softmax**: Converts raw output scores into probabilities, Often used in the output layer of multi-class classification models and, more recently, in the implementation of attention mechanisms [21]. This function is defined mathematically as:

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}} \quad (2.5)$$

The training process of a neural network involves iteratively adjusting its weights and biases to minimise errors, allowing the network to accurately model data [21]. This process is carried out through the following series of steps to refine these parameters and improve the network performance:

- **Forward Propagation**: Input data is processed sequentially through the layers, with each neuron applying transformations to produce predictions at the output layer [21].
- **Loss Function**: The difference between the predicted output of the network and the true labels is measured using a loss function. Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy Loss for classification [21].
- **Backpropagation**: This algorithm computes the gradients of the loss function with respect to each trainable parameter (weights and biases) in the network. It relies on the chain rule of calculus to propagate errors backward through the network [46].
- **Optimisation Algorithms**: An optimiser updates the weights and biases based on the computed gradients with the objective of reducing the loss. Some examples of optimisation algorithms include Stochastic Gradient Descent (SGD) and adaptive optimisers like Adaptive Moment Estimation (Adam) [31].

2.1.2 Learning Mechanisms in Neural Networks

The learning mechanism of neural networks is the set of methods and processes through which a network learns from data, adjusts its parameters, and enhances its performance [21]. The primary modes of learning in neural networks are supervised, unsupervised, semi-supervised, self-supervised, and reinforcement learning [2, 45]. Each of these learning paradigms differs in how they handle data, their training strategies, and the types of problems they solve. Below, are the fundamental characteristics, methodologies, and applications of each learning paradigm.

Supervised Learning

Supervised learning is the most commonly used type of learning in neural networks. In this approach, the model is trained on labeled data, meaning each input in the training dataset is paired with a corresponding correct output (also known as *label*). The goal of supervised learning is to learn a mapping function from inputs to outputs such that the network can accurately predict outcomes for unseen data [33].

During training, the network uses the provided labels to adjust its weights and biases by minimising the error or loss, which is the difference between the predicted outputs and the actual labels. This is typically done through the processes of forward propagation, loss calculation, and backpropagation, as outlined earlier in the training process.

Common applications of supervised learning include classification tasks, such as image recognition and speech recognition, as well as regression tasks, such as predicting numerical values for financial forecasting. These applications demonstrate the versatility of supervised learning in handling both categorical and continuous data across various domains.

Unsupervised Learning

Unsupervised learning differs from supervised learning as it does not rely on labelled output data. Instead, the network is given raw input data without explicit labels and must identify patterns and relationships on its own [6]. The primary objective of unsupervised learning is to discover the underlying structure in the data and extract meaningful representations from it.

A fundamental technique used in unsupervised learning is clustering, where the model groups similar data points together [30]. K-means clustering and hierarchical clustering are common methods used for this purpose, where the network iteratively assigns data points to clusters based on their similarities. Another approach is principal component analysis (PCA), which reduces the dimensionality of the data, by means of compression, while retaining important features.

Unsupervised learning is useful in scenarios where labelled data is scarce or unavailable, such as when working with large datasets or data with unknown labels. Tasks like anomaly detection, customer segmentation, and feature extraction, which require identifying hidden patterns and structures in data are common applications of unsupervised learning.

Semi-Supervised Learning

Semi-supervised learning combines elements of both supervised and unsupervised learning by utilising a small amount of labelled data alongside a larger pool of unlabelled data. The labelled examples guide the model, while the unlabelled data helps uncover additional patterns and structures [9, 30].

This mechanism is a practical solution for domains with vast amounts of data but limited labelled resources. Combining a small set of labelled data with a much larger pool of unlabelled data, semi-supervised learning allows models to learn effectively and achieve high accuracy while significantly reducing the need for extensive human interaction.

This approach is especially useful in fields where labelling data is expensive or time-consuming, such as in medical imaging and Natural Language Processing (NLP).

Self-Supervised Learning

Self-supervised learning is a subset of supervised learning where the model generates its own labels from unlabelled data, instead of using manually annotated datasets. This model learns by solving pretext tasks designed to help it recognise meaningful patterns. The goal of this pre-training phase is to equip the model with transferable knowledge that can be applied to similar tasks while using minimal labeled data [45].

Self-supervised learning makes it possible to train models using vast amounts of unlabelled data, significantly reducing the need for human annotation while also improving model generalisation.

This approach is particularly useful for tasks where obtaining labeled data is expensive or impractical, such as NLP. Language models like BERT and ChatGPT rely on this learning mechanism to understand context and generate human-like text.

Reinforcement Learning

RL represents a type of learning mechanism focused on sequential decision-making tasks [57]. In RL, an agent interacts with its environment, making decisions and taking actions based on its current state. As the agent makes those actions, it receives feedback in the form of rewards or penalties, which is used to inform its future actions.

The main feature of RL is the concept of exploration and exploitation. The agent explores different actions to discover those that lead to higher rewards while exploiting known strategies to maximise rewards in the short term. Over time, the agent learns an optimal policy that maximises cumulative rewards through trial and error.

Reinforcement learning is applied across a variety of domains, such as gaming where the use of AI provides bigger challenges for players, robotics where machines are trained to perform complex tasks like object manipulation and autonomous navigation, and finance where algorithms optimise trading strategies based on market conditions.

2.2 Deep Learning

Deep Learning (DL) is a specialised subset of machine learning that utilises neural networks with multiple interconnected layers, known as Deep Neural Networks (DNNs), to automatically discover features and representations within large volumes of data [21]. By using architectures with increased depth and complexity, deep learning models are capable of extracting patterns directly from raw input data, often surpassing the capabilities of traditional machine learning techniques.

This section provides an overview of the principles of deep learning, its evolution from traditional machine learning approaches, and an examination of commonly used architectures.

2.2.1 Foundations of Deep Learning

The journey from neural networks to deep learning began with the introduction of the perceptron model in the 1940s, inspired by biological neural systems. However, these early networks were limited by their basic architectures and inability to solve complex, non-linear problems [14].

The creation of deep learning in the late 2000s was possible due to the broader availability of larger scale datasets, the increased computational power through the use of Graphics Processing Units (GPUs), and the improvement of already existing algorithms, by making use of better activation functions and optimisation techniques.

Deep learning distinguishes itself from traditional neural networks through the use of deep architectures characterised by multiple layers. The term "deep" refers to the depth (number of layers in the neural network), which allows the model to learn hierarchical features from raw data, making them ideal for tasks involving unstructured data such as images, text, and speech [21].

These advances in data availability, computational resources, and algorithmic innovations have transformed deep learning into a cornerstone of modern artificial intelligence.

2.2.2 Common Deep Learning Architectures

Deep learning architectures are designed to address different types of data and tasks by leveraging specialised network structures [21]. Each architecture has unique characteristics that make it optimal for specific applications. Some commonly used deep learning architectures include Feedforward Neural Networks (FNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Transformer Networks, and Generative Adversarial Networks (GANs).

Feedforward Neural Networks (FNNs) are the simplest type of artificial neural networks, where information flows in one direction, going from input to output, without cycles or loops [35]. These networks consist of an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to neurons in the next layer through weighted connections, and activation functions introduce non-linearity to enable complex pattern learning. FNNs are commonly used for tasks such as classification and regression in structured data, but they lack

the ability to capture spatial or sequential dependencies.

CNNs are specifically designed for processing grid data structures such as images and videos. CNNs utilise convolutional layers that scan small patches of the input and apply filters to detect spatial features such as edges, textures, and shapes [2, 21]. These layers help preserve spatial relationships in the data while reducing the number of parameters, making CNNs more efficient for larger-scale image processing tasks. Convolutional architectures exploit weight sharing and local connectivity, which improve computational efficiency and allow the model to capture hierarchical feature representations, from low-level edges in the first layers to high-level semantic concepts in deeper layers. CNN architectures typically consist of the following types of layers:

- **Convolutional:** Layers that apply learnable filters (trained to identify various patterns or objects within data) to the input, to extract meaningful features.
- **Downsampling Layers:** Layers that reduce the spatial dimensions of feature maps, often using pooling to select the most important values in local regions, improving computational efficiency and reducing sensitivity to small input variations.
- **Upsampling Layers:** Layers that increase the spatial dimensions of feature maps.
- **Fully Connected:** Layers that integrate the extracted features and map them to the final output.

The complexity of CNNs depends on factors such as the depth of the network, the number of filters per layer, and the size of these filters. These characteristics directly influence both the representational capacity and the computational cost of the model. Smaller architectures with fewer filters may be more efficient but risk underfitting complex data, while deeper networks with many filters can capture rich hierarchical features at the expense of increased memory usage and inference time [21].

CNNs are commonly used for tasks such as image classification, object detection, and medical imaging, where capturing spatial hierarchies in data is required [2, 33]. The accurate feature extraction provided by CNNs is also essential for applications with facial recognition features and autonomous driving capabilities.

Recurrent Neural Networks (RNNs) are designed to process sequential data by maintaining a memory of previous inputs through recurrent connections [2, 21]. RNNs incorporate loops that allow information to persist across time steps, making them suitable for tasks involving temporal dependencies.

RNNs share parameters across different time steps, allowing them to recognise patterns in sequences of varying lengths. This structure allows them to capture context and relationships within sequential data, making them useful for tasks where order and history are important.

This type of network is widely used in applications that require sequential processing, such as NLP and speech recognition. RNNs are usually applied for tasks like machine translation, text generation, and capturing contextual relationships across words and phrases in a sequence.

Generative Adversarial Networks (GANs) are a class of deep learning models designed for generative tasks, where the goal is to create new data samples that resemble a given dataset. GANs consists of two neural networks, the generator and the discriminator, that are trained at the same time competing against each other in a process known as adversarial training [10]. The generator is responsible for producing synthetic data samples, attempting to generate outputs that closely resemble real data. The discriminator acts as a classifier that differentiates between real and generated samples. During training, the generator aims to fool the discriminator by improving the quality of its outputs, while the discriminator continuously refines its ability to distinguish real from fake data. This adversarial process drives both networks to improve over time, leading to the generation of highly realistic data samples.

Due to their ability to generate high-quality synthetic data, GANs have been widely used in various domains, including image synthesis, data augmentation, realistic human face generation, and the enhancement of low-resolution images.

Transformer Networks represent a breakthrough in deep learning, particularly in the field of NLP. Transformers use a self-attention mechanism that allows them to analyse entire input sequences simultaneously [1]. This parallel processing capability significantly enhances efficiency and enables the model to capture broader dependencies more effectively. Transformers are widely used in tasks such as machine translation, text generation, and speech recognition. Their versatility has also led to their adoption in fields like computer vision and reinforcement learning, where they have demonstrated strong performance in handling complex patterns and dependencies.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a subset of machine learning that focuses on how agents learn to make decisions by interacting with an environment and observing the consequences of their actions [21, 40]. RL relies on the concept of trial and error, where an agent learns by receiving feedback in the form of rewards or penalties. This characteristic allows RL to address sequential decision-making and control problems across a variety of domains.

This section explores the fundamental principles and components of RL, discusses its key algorithms, and highlights some of its real-world applications.

2.3.1 Fundamentals of Reinforcement Learning

Reinforcement Learning (RL) is based on the interaction between an agent and its environment, where the goal is to learn an optimal policy to make decisions [40]. RL focuses on sequential decision-making, where actions influence not only immediate rewards but also future outcomes. Through numerous repeated interactions, the agent refines its behaviour to maximise cumulative rewards over time.

The RL framework is depicted as a cycle of perception, decision-making, and learning. The agent observes the current state of the environment, selects an action based on its policy,

and receives feedback in the form of a reward (as shown in Figure 2.2). This feedback helps the agent adjust its policy to improve future performance.

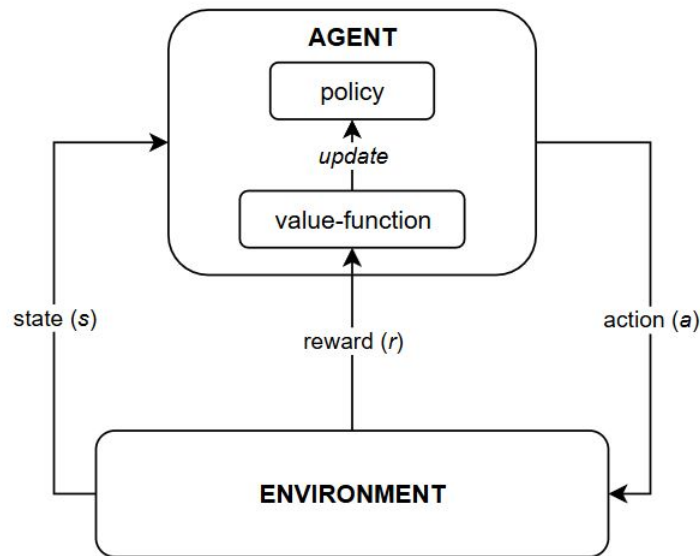


Figure 2.2 Reinforcement Learning Scheme

This learning mechanism allows an agent to develop intelligent behaviors in dynamic environments by relying on the following key components[40]:

- **Agent:** The entity that learns and makes decisions based on interactions with the environment.
- **Environment:** The external system in which the agent operates, defining the rules of interaction and determining the consequences of the agent's actions.
- **State (s):** A representation of the environment at a given time, containing all the necessary information the agent needs to make an informed decision.
- **Action (a):** The set of possible choices available to the agent that change the state of the environment in some way.
- **Reward (r):** The numerical value indicating the immediate outcome of an action. The agent's objective is to maximise the cumulative reward over time.
- **Policy (π):** The strategy that dictates the agent's actions based on the current state.
- **Value Function (V):** A measure of the desirability of a state, considering both immediate rewards and expected future rewards which helps the agent evaluate the quality of different states beyond short-term gains.

To better understand the logic behind the interaction scheme depicted above (Figure 2.2), we can consider a gaming scenario where an RL agent is tasked with solving a maze. In

this setting, the agent continuously observes its current location and the overall layout of the maze. Based on this information, it makes decisions, such as choosing which direction to move, with the objective of finding the exit. After each action, the agent receives feedback in the form of rewards that indicate whether it is moving closer to or farther from the goal or if it has encountered any obstacle or dead-end. This feedback, combined with an evaluation of the long-term benefits of being in particular states (through a value function), drives the agent to refine its strategy over time. As the agent gains more experience through repeated interactions with the maze, it gradually improves its decision-making process to maximise cumulative rewards.

2.3.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an advanced paradigm that combines the decision-making framework of RL with the powerful representation capabilities of DL [39, 40]. Deep RL employs deep neural networks to approximate value functions, policies, and models of the environment, directly from raw data. Deep RL algorithms can be categorised into two main approaches (Figure 2.3), model-free and model-based learning.

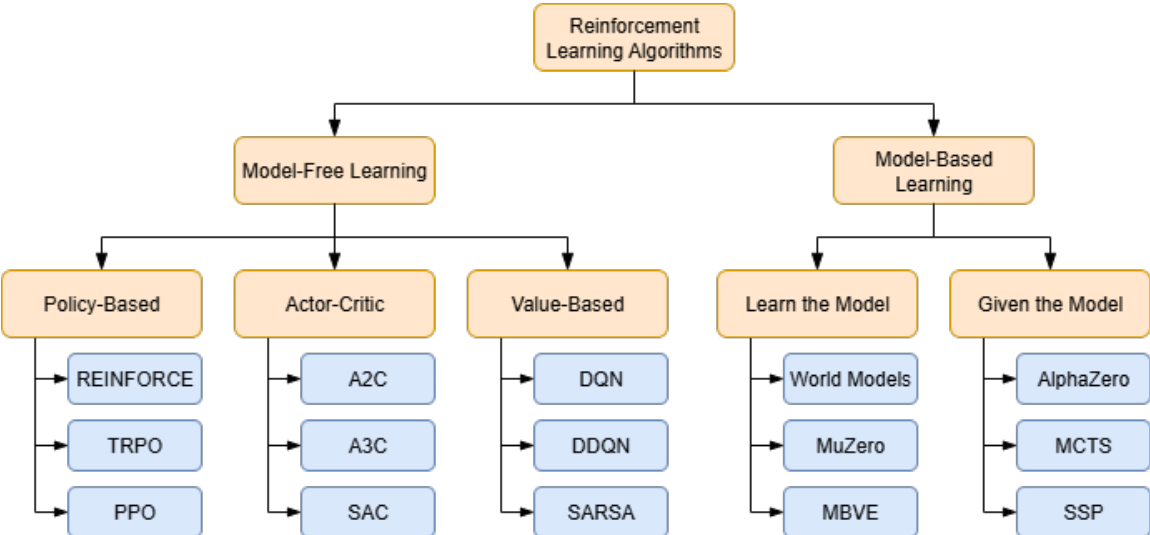


Figure 2.3 Deep Reinforcement Learning Algorithm Taxonomy

Model-based algorithms attempt to learn a model of the environment’s dynamics, predicting the next states, and rewards given the current states and actions. Once the model is fully learnt, it can be used to plan and determine optimal actions. Although more sample-efficient, these methods are often less stable, due to wrong model predictions in more complex environments. Model-based approaches can be categorised into two types: those that learn the model from data (learning-based), such as World Models [22], MuZero [50] and Model-Based Value Expansion (MBVE) [16], those where the model was provided beforehand, such as AlphaZero [53], Monte Carlo Tree Search (MCTS) [59] and Stochastic Shortest Path (SSP) [61].

Model-Free algorithms do not attempt to model the environment, directly learning policies or value functions from experiences instead. These methods are usually more robust but require larger amounts of data. Model-free algorithms are divided into three methods, policy-

based, value-based, and actor-critic methods, each with distinct strategies for learning and decision making.

Policy-based methods, such as REINFORCE [63], Trust Region Policy Optimization (TRPO) [51] and PPO [52], directly optimise the policy $\pi(a|s)$ by maximising expected rewards. These methods are more flexible and particularly effective in high-dimensional or continuous action spaces. They can better handle multi-modal action distributions and avoid the need for value maximisation steps. These approaches often suffer from high variance in gradient estimates, which can lead to slower convergence and poor sample efficiency when rewards are sparse or delayed [5].

Value-based methods, such as DQN [38], DDQN [25] and State-Action-Reward-State-Action (SARSA) [47], focus on learning value functions that estimate the expected return of taking specific actions in a given state. These methods often use techniques like experience replay and target networks to stabilise learning and improve sample efficiency. Value-based approaches are generally well-suited for environments with discrete and low-dimensional action spaces, achieving strong training performances and high inference efficiency with relatively simple architectures. However, these methods tend to struggle in continuous action spaces and are sensitive to hyperparameter tuning and reward sparsity [40].

Actor-critic methods, such as Advantage Actor-Critic (A2C) [37], Asynchronous Advantage Actor-Critic (A3C) [37] and Soft Actor-Critic (SAC) [23], combine value-based and policy-based approaches by maintaining both a policy (actor) and a value function (critic). The critic guides the actor's updates, leading to more stable and efficient learning. This hybrid structure benefits from the reduced variance provided by the critic, while retaining the flexibility of direct policy optimisation. Although more complex to implement and tune, actor-critic methods offer a practical balance between performance and stability [40].

These methods, while enabling significant advances in reinforcement learning, often require large amounts of training data, can be unstable, and are highly sensitive to hyperparameters. Therefore, optimisation techniques are necessary to improve learning efficiency, training stability, and overall robustness.

2.3.3 Deep Reinforcement Learning Enhancements

Deep Reinforcement Learning algorithms face several challenges that limit their efficiency and stability, such as high training variance, poor sample efficiency, unstable convergence, and sensitivity to hyperparameters. To address these issues, a variety of enhancements have been implemented to improve learning performance, exploration, and provide more accurate value estimates. These enhancements include: Double Q-learning, Prioritised Experience Replay, Dueling Network Architectures, Multi-step Returns, Distributional Value Learning, and Noisy Networks.

Double Q-learning [25] was introduced as an improvement over the Deep Q-Networks algorithm, which suffered from a tendency to overestimate action values due to the use of the same network for both action selection and evaluation. Double Q-learning addresses this

by splitting these two roles, using one network to select the actions and a second separate network to evaluate them. This enhancement reduces overestimation bias, providing more accurate value estimates and results in improved stability and performance.

Prioritized Experience Replay (PER) [48] improves the standard replay buffer by recognising that not all past experiences are equally useful for learning. This ensures that experiences with higher Temporal-difference (TD) error, which carry more informative updates, are replayed more frequently, increasing sample efficiency. Instead of sampling uniformly, PER assigns higher sampling probability to transitions with greater learning potential, typically measured using the magnitude of the TD error. The probability of sampling a transition i is then given by:

$$P(i) = \frac{|\delta_i|^\alpha}{\sum_k |\delta_k|^\alpha} \quad (2.6)$$

where δ_i is the TD error for transition i , and α determines how strongly prioritisation is applied. This mechanism ensures that updates are focused on the most informative experiences, accelerating the algorithm's convergence and improving sample efficiency, at the cost of additional computational resources.

Dueling Network Architectures [62] are architectural improvements to value-based networks, accomplished by decomposing the Q-value function into two separate streams (one estimating the state value function $V(s)$ and another estimating the advantage function $A(s, a)$). These streams are then combined to produce the Q-value:

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \right) \quad (2.7)$$

This improved design allowed the network to learn more effectively which states are valuable regardless of the chosen action, while still distinguishing which actions are advantageous within those states, improving generalisation and stability in environments where many actions have similar consequences.

Multi-step Returns [11] enhance the traditional one-step temporal-difference updates by considering multiple future rewards. Instead of updating the value estimate after a single reward, the algorithm accumulates rewards over n steps before backing up a value estimate. This enhancement provides a richer learning signal, which helps the agent spread its reward information more effectively across states, resulting in a faster and less biased learning process.

Distributional Value Learning [4] is a redefinition of the traditional value functions, achieved by modelling not just the expected return, but the entire distribution of possible returns for a given state-action pair. This shift from scalar to distributional representation allows the algorithm to capture uncertainty more accurately and to use richer learning signals, improving performance in complex domains such as Atari games.

Noisy Networks [17] are an alternative to exploration strategies such as ϵ -greedy by adding learnable noise directly into the network's weights. Instead of relying on fixed randomness in action selection, the network itself learns when and how to explore by adjusting the scale of the injected noise during training. This approach allows exploration to be adaptive and

state-dependent, enabling agents to discover more effective behaviours without relying on fixed exploration schedules.

These techniques represent significant enhancements to the foundations of DRL, improving algorithm efficiency, stability, exploration, and overall performance. Although each method only tackles a specific flaw or weakness, they can usually be combined to produce stronger and more robust algorithms, substantially expanding the capabilities of DRL.

2.3.4 Applications of Reinforcement Learning

Reinforcement Learning (RL) has a variety of applications due to its ability to learn optimal decision-making strategies through trial and error. This ability to improve performance through interaction with dynamic environments makes RL a powerful tool across various fields, such as robotics, NLP, and gaming [40].

In robotics, RL is used to enable robots to learn tasks such as object manipulation and navigation. By interacting with their environment and receiving feedback, robots can continuously improve their skills and adapt to new situations without being explicitly programmed for each task.

In NLP, tasks such as dialogue systems, machine translation, and summarisation are enhanced with the use of RL. By treating language generation as a sequential decision-making problem, RL helps models optimise responses based on long-term objectives, such as conversational flow or the accuracy of translations.

Gaming has become one of the most prominent areas for RL applications. Agents have demonstrated remarkable success in mastering complex games, where they learn to develop optimal strategies through trial and error. These agents can navigate highly dynamic environments, adapting to changes in game state and making decisions that maximise long-term rewards.

2.4 Deep Learning in Embedded Systems

Deep learning is usually associated with high-performance computing environments that rely on powerful GPUs and cloud-based resources. However, recent advancements in hardware and software optimisation techniques have enabled the deployment of deep learning models in embedded systems. This integration expands the applicability of deep learning beyond traditional data centres, allowing AI-driven functionalities in real-time, low-power, and resource-constrained environments [36].

In this section, we explore the main challenges of deploying deep learning on embedded systems, discuss optimisation techniques that enable efficient model execution, and examine real-world applications where deep learning is successfully integrated.

2.4.1 Challenges of Deploying Deep Learning on Embedded Systems

The transition from traditional deep learning environments to embedded systems introduced several challenges, mainly due to the lack of resources on these devices. Unlike high-performance computing setups, which relied on modern GPUs and vast memory resources, embedded systems must balance computational efficiency, power consumption, and real-time processing capabilities [28, 36].

As deep learning models became more complex, their integration into embedded systems required significant adaptation to ensure viability and efficiency and overcome the main challenges associated with deploying deep learning models on embedded systems. These challenges included:

- **Computational Constraints:** Embedded devices usually have limited processing power, making it difficult to run large neural networks efficiently.
- **Memory Limitations:** Storing and loading deep learning models requires significant memory, which is often scarce in embedded systems.
- **Energy Efficiency:** Many embedded applications operate on battery power, which requires deep learning models with minimal energy consumption.
- **Latency Requirements:** Certain applications, require fast inference speeds, which can be difficult to achieve due to the inherent latency of embedded systems.
- **Thermal Constraints:** The power consumption of deep learning models leads to heat generation, which must be managed effectively to prevent performance degradation or hardware failure.
- **Security and Privacy:** Embedded systems typically process sensitive data locally, which increases the risk of unauthorised access and data breaches.

In order to overcome these challenges and ensure the effectiveness of deep learning models in embedded environments, a variety of techniques have been developed. These methods aim to reduce computational and memory requirements while maintaining the performance and accuracy of deep learning models.

2.4.2 Techniques for Deploying Deep Learning in Embedded Systems

Deploying deep learning models on embedded systems requires addressing the various challenges associated with limited resources. To address these constraints, several techniques have been developed, including model compression and pruning, knowledge distillation, efficient neural network architectures, and edge computing with hardware acceleration. These methods enhance the efficiency of deep learning models, reducing computational demands while preserving their effectiveness.

Model Compression and Pruning [24] is a technique that aims to reduce the size of neural networks, making them more manageable for embedded systems. This is achieved through methods like weight pruning (the removal of redundant or less important connections in a neural network), resulting in a smaller and more efficient model with a minimal loss in accuracy. Another method used to achieve this is quantisation, which reduces the precision of the model weights and activations. By lowering the bit-width (for example, from 32-bit floats to 8-bit integers), quantisation reduces the computational load and memory requirements, making it easier to deploy deep learning models on hardware with limited resources.

Knowledge Distillation [64] is a technique that involves transferring knowledge from a large, pre-trained model (called the teacher) to a smaller model (called the student). The teacher model, with its high capacity, is used to train the smaller student model, which can be deployed more easily on embedded systems. The student model retains much of the accuracy of the teacher but with fewer parameters and a reduced computational footprint. This method allows for the effective transfer of knowledge, enabling smaller models to learn from more powerful, computationally expensive networks.

Efficient Neural Network Architectures [32] are specialised architectures designed to be lightweight while maintaining strong performance on tasks such as image classification and object detection. These architectures use techniques, such as depthwise separable convolutions or fire modules, to reduce the number of parameters and computations needed for processing, leading to models running faster and consuming less power.

Finally, Edge Computing and Hardware Acceleration [8] provide powerful solutions to improve the performance of deep learning models in embedded systems. Edge computing allows data to be processed locally on embedded devices rather than being sent to the cloud, reducing latency and minimising the need for constant network communication. Hardware accelerators are designed to perform matrix operations and other computations efficiently, significantly speeding up the execution of neural networks and enabling embedded systems to run deep learning models in real-time even with limited resources.

These techniques allow deep learning models to be efficiently deployed on embedded systems, overcoming the challenges posed by limited resources. In many cases, multiple techniques are used together to achieve an optimal balance between performance, energy efficiency, and computational cost.

2.4.3 Applications of Deep Learning in Embedded Systems

The integration of deep learning into embedded systems has opened new possibilities across a wide range of fields. By enabling intelligent data processing on resource-constrained devices, deep learning allows for real-time decision-making without relying on constant connectivity.

Autonomous vehicles and driver assistance technologies make use of embedded deep learning models to process data from cameras and sensors to recognise objects, detect lane markings, and anticipate potential dangers. Techniques like CNNs are used for real-time image recognition, while RNNs and RL enable predictive decision-making for navigation and obstacle

avoidance [13].

The increased use of IoT devices has benefited significantly from advances in deep learning. Embedded deep learning enables smart home assistants, voice recognition systems, and personalised user experiences in consumer electronics [54]. Devices like smart thermostats, refrigerators, and fitness trackers use deep learning models to learn user behavior and optimise performance accordingly.

Finally, the gaming industry has been revolutionised by deep learning, by enhancing AI-driven interactions, graphics, and user experiences with embedded systems. Advanced AI models enable NPCs to learn and adapt to player behavior in real-time, increasing immersion. Embedded AI also enhances game controllers and Virtual Reality (VR) devices, enabling gesture recognition, voice commands, and adaptive difficulty adjustments for a more personalised user experience.

The integration of deep learning into embedded systems has expanded the capabilities of various industries. As hardware advancements continue and optimisation techniques evolve, deep learning applications in embedded environments will become even more efficient and widespread, driving innovation across multiple domains.

2.5 Algorithm Optimisation

Training and deploying deep learning models require significant computational resources, making optimisation a fundamental aspect of deep learning research. Algorithm optimisation techniques aim to enhance the efficiency, accuracy, and scalability of neural networks while reducing their computational cost.

These optimisation methods involve trade-offs that must be carefully managed to ensure optimal performance [26]. Reducing model size and complexity can improve efficiency, but may lead to a decline in accuracy. Techniques that accelerate training can introduce instability, affecting convergence. Methods that enhance generalisation to unseen data often require greater computational resources, increasing overall costs.

Achieving a balance between these factors is essential for developing models that are both effective and practical for real-world applications. To address these challenges, various optimisation techniques have been proposed to enhance performance while mitigating trade-offs.

2.5.1 Optimisation Techniques for Deep Learning Algorithms

In deep learning, numerous optimisation techniques have been developed to improve training speed, generalisation performance, and computational efficiency. These methods address the challenges imposed by complex models and massive datasets.

Hyperparameter tuning is a fundamental optimisation technique that focuses on selecting the most effective values for parameters such as learning rates, batch sizes, network depths, and activation functions. The tuning process can be carried out through manual trial-and-

error or by relying on commonly recommended values for each algorithm. Proper tuning can significantly improve training stability, convergence speed, and overall model performance.

Gradient-based optimisation algorithms are fundamental to deep learning training. Stochastic Gradient Descent (SGD) updates model parameters incrementally based on the gradient of the loss function, making it efficient for large-scale problems [7]. Adaptive Moment Estimation (Adam) builds on this by incorporating both momentum and adaptive learning rates, leading to faster convergence and improved stability [31]. RMSprop dynamically adjusts learning rates during training, helping stabilise convergence, especially in recurrent neural networks [66].

Regularisation techniques help prevent overfitting and improve generalisation, ensuring that models work efficiently on unseen data without excessive complexity. Dropout randomly deactivates neurons during training, reducing co-adaptation and increasing model robustness [56]. L1 and L2 regularisation, commonly known as weight decay, imposes penalties on large weight values, encouraging the network to learn simpler and more generalisable patterns [42].

Normalisation methods such as batch normalisation and layer normalisation further stabilise and accelerate the training process. Batch normalisation normalises the activations within each mini-batch, reducing internal covariate shift and allowing higher learning rates [29]. Layer normalisation normalises the features of a single instance, ensuring consistent performance under different training conditions [3].

Parallel and distributed training techniques are beneficial for large-scale models and datasets. Data parallelism distributes workload across multiple GPUs or TPUs, accelerating training by processing large datasets simultaneously, while model parallelism partitions a large network across several devices to overcome memory limitations [12].

Neural Architecture Search (NAS) automates the design of deep learning architectures by exploring the architectural design space, identifying configurations that optimise both performance and efficiency, outperform manually designed models, and reduce the time and expertise needed for network design [68].

These optimisation techniques can be integrated to enhance algorithms for both research and practical applications, enabling efficient training and improved performance even when addressing the challenges posed by complex models and larger-scale data.

2.6 Related Work

Several studies have examined the implementation and optimisation of RL and DRL algorithms, providing valuable insights into improving their efficiency, scalability, and deployment in constrained environments.

This section provides an overview of key contributions in the field, highlighting notable advancements, implementation strategies, and practical applications of deploying RL and DRL in resource-constrained environments.

2.6.1 Reinforcement Learning Algorithms and Performance Comparison

The field of Reinforcement Learning is continuously evolving, with new algorithms and improvements being regularly proposed to address existing limitations and improve performance. These new methods are typically evaluated by comparing them against established baselines across a range of benchmark tasks. Since different methods often present contrasting trade-offs in terms of stability, efficiency, and computational requirements, reviewing examples from various approaches helps establish a solid foundation for understanding their capabilities and limitations, as well as for identifying suitable candidates for further experimentation.

Schulman et al. (2017) [52] introduced Proximal Policy Optimisation (PPO), a policy-based method designed to improve the stability of policy-gradient updates while maintaining a straightforward implementation. PPO achieves this by employing a clipped surrogate objective, which constrains changes to the policy between updates and prevents overly large steps that could destabilise training. Experimental results demonstrated that PPO delivers superior performance compared to previous methods such as TRPO and Efficient Actor-Critic with Experience Replay (ACER) across a wide range of tasks. According to the authors, this algorithm is simpler, more robust, and provides a more stable learning behaviour, making it a strong baseline for reinforcement learning research. However, being an on-policy method, PPO can be less sample-efficient and more computationally demanding in scenarios with long training sessions, which may limit its practicality in resource-constrained environments.

Hessel et al. (2017) [27] proposed Rainbow, a value-based algorithm that integrates several enhancements to DQN into a single framework. These enhancements include double Q-learning, PER, duelling network architectures, multi-step returns, distributional value learning, and noisy networks. By comparing their algorithm with the available Atari benchmarks, the authors demonstrated that Rainbow achieves cutting-edge performance, with studies showing that each component contributes to its overall success. Rainbow's strengths lie in its improved data efficiency and its ability to leverage complementary techniques for significant performance gains in discrete action spaces. However, the combination of multiple components increases both the implementation complexity and the computational load, potentially making it less suited for resource-constrained devices where memory and processing resources are limited.

Although both algorithms have demonstrated strong performance on standard benchmarks, their original evaluations focused primarily on metrics such as the highest scores achieved,

without accounting for the time required to obtain them, and on learning curves measured under unconstrained computing conditions. Both algorithms will be considered as potential candidates for experimentation, but the comparative analysis in this work will differ from the original studies by incorporating metrics that reflect the constraints of embedded hardware, such as execution times, Central Processing Unit (CPU), GPU and memory usage, as well as the overall complexity and suitability of the model for deployment on embedded devices. This perspective aims to identify algorithms that not only achieve high in-game performance but also operate efficiently within the strict limitations of embedded systems.

2.6.2 Reinforcement Learning in Gaming

Reinforcement Learning (RL) has significantly influenced game AI, transforming the way agents learn and adapt to complex environments. Traditional game AI techniques, such as finite state machines and behaviour trees, offer structured but rigid responses, lacking the adaptability needed for dynamic gameplay. Utilising RL based approaches enables AI to learn optimal strategies through experience, allowing more lifelike and unpredictable Non Playable Characters (NPCs) and engaging player interactions.

Souchleris et al. (2023) [55] provides a comprehensive review of RL applications in the gaming industry, emphasising its potential for enhancing game AI, procedural content generation, and player experience. They highlight how RL agents can dynamically adjust their strategies based on player behaviour, making NPCs more immersive. However, this approach has increased computational costs associated with training RL agents, as high-performance hardware and extensive simulations are often required. To mitigate these issues, the authors suggest hybrid approaches that integrate RL with rule-based methods or supervised learning, which can improve efficiency while maintaining adaptability.

Building on this foundation, Sá and Madeira (2024) [60] specifically focused on the use of DRL in Real-time Strategy (RTS) games, where decision-making must occur under strict time constraints and high-dimensional action spaces, since RTS games require agents to control multiple units, manage resources, and react to unpredictable opponents in real-time. The authors discuss several techniques, such as hierarchical reinforcement learning and multi-agent reinforcement learning, which help decompose these complex decision-making processes into more manageable sub-problems. The main challenge present in this type of environment is opponent modelling, where agents must anticipate and counter enemy strategies. RNNs and attention mechanisms have been explored to enhance adaptability, allowing AI to dynamically respond to opponents' actions. The authors conclude by emphasising that training efficiency remains a challenge and suggest that future research should also explore hybrid approaches, such as combining RL with symbolic reasoning and leveraging transfer learning techniques to enable AI models trained in one RTS game to be applied to others.

Both articles explore the broader applications of RL in gaming AI. However, given that this thesis focuses primarily on Atari games, it is necessary to ensure that the developed models can generalise effectively across different game environments. Jiajun Fan (2023) [15] ad-

dresses this concern, by providing a comprehensive analysis of DRL applications within the Atari domain, which has become a standard benchmark for evaluating RL algorithms. This article criticises existing evaluation metrics, where many RL agents often exploit game mechanics instead of developing genuine strategic understandings. To address this, the author proposes benchmarking RL performance against human world records instead of average human play. The limited generalisation ability of DRL models is another point made by the authors, where some models often excel in specific Atari games but struggle to transfer learned strategies to new environments due to reliance on pixel-based inputs and reward shaping. This study also highlights some model issues, such as the trade-off between exploration and exploitation, where agents struggle to explore effectively while maximising rewards, leading to suboptimal learning in sparse reward environments, and the lack of sample efficiency, as training DRL models often require millions of frames, making large-scale real-world applications impractical. To improve scalability and adaptability, the author suggests solutions like meta-learning, incorporating diverse training environments, and utilising world models that enable agents to predict future states before acting.

2.6.3 On-device Reinforcement Learning

Beyond traditional gaming applications, deploying RL models on embedded systems requires addressing many resource constraints such as limited processing power, memory, and energy. Running complex RL algorithms efficiently in these environments is challenging and requires optimising frameworks to improve performance, stability, and resource efficiency.

These challenges were tackled in Li et al. (2023) [34] by proposing R^3 , an on-device RL framework designed for autonomous robotics. Their approach emphasises real-time responsiveness, leveraging dynamic batch-size adjustment, efficient memory management, and adaptive resource coordination to optimise model performance. Unlike traditional RL systems that might rely on external servers for training, R^3 enables learning directly on embedded hardware, reducing latency and increasing autonomy.

A broader perspective on on-device training is provided by Zhu et al. (2024) [67] which reviews existing frameworks for federated learning, model quantisation, and adaptive training in resource-constrained environments. The authors explore the trade-offs between local computation and communication overhead, an issue particularly relevant for decentralised environments. While not strictly focused on RL, this article offers some knowledge on how incremental learning techniques can be used for embedded RL systems that need real-time adaptation without relying on external servers.

Since hardware constraints are the main challenge present in most studies, there is a need to optimise RL models for limited resource environments. Svoboda et al. (2021) [58] explores the implementation of RL models for ultra-low-power microcontrollers, a domain often referred to as TinyML. Their research introduces techniques such as reward shaping and hybrid learning paradigms to optimise RL models for minimal memory usage and power consumption, specifically targeting devices with strict resource limitations. This type of optimisation

will be necessary for most embedded devices running DRL models, including applications for autonomous game playing, such as the one that will be developed in this thesis.

While the mentioned studies provide valuable insights into various aspects of DRL implementation, a notable gap remains in research that addresses the unique challenges of deploying autonomous game-playing agents on embedded systems. Most existing works either explore DRL applications in gaming without considering hardware constraints or address embedded system implementations without accounting for the specific complexities of dynamic game environments. This work seeks to fill this gap by developing an efficient, on-device learning approach that not only adapts to the resource limitations of embedded systems, but also fulfils the intricate demands of real-time gaming scenarios.

2.7 Chapter Summary

This chapter established the theoretical fundamentals necessary to understand the thesis. It began by detailing the core concepts of Neural Networks (NN) and their evolution into DL, explaining the essential architectural basis. Following this, the principles of RL were introduced, leading to a comprehensive exploration of DRL, highlighting its key components, advanced algorithms and the challenges of deploying these models on resource-constrained embedded systems, which necessitated a review of various algorithm optimisation techniques.

The chapter also summarised the related work by reviewing existing applications of DRL, particularly in the domain of autonomous game playing, as well as current research into algorithm deployment on resource-limited hardware. This review of existing literature served to pinpoint the lack of comprehensive research addressing the simultaneous challenges of developing and deploying a DRL agent within the resource limitations of embedded systems.

This theoretical foundation and contextual background paved the way for the practical implementation phase. The next chapter will detail the systematic methodology for the design, implementation, evaluation, and optimisation of selected DRL algorithms, developed to address the constraints of embedded systems.

Chapter 3

Analysis and Evaluation of Deep Reinforcement Learning Algorithms for Embedded Game Playing

The purpose of this chapter is to present the practical implementation of the system, based on the concepts and methodologies previously discussed in the background and related work. It outlines the transition from theoretical foundations to applied development, describing all the tools, frameworks, and techniques used to construct and evaluate the DRL algorithms implemented in this project.

This chapter traces the project from its initial requirements and design decisions to the implementation and preliminary evaluation processes. It begins by outlining the objectives and constraints that guided the design of the system, followed by the framework and approaches adopted to structure development. The focus then shifts to the practical implementation, where environments, algorithms, and optimisation strategies were implemented and preliminary tested to adapt them for embedded devices. The structure of this chapter is as follows:

- **Section 3.1** introduces the Design Space Exploration framework, outlining the proposed methodology, project requirements, and design framework.
- **Section 3.2** details the process of exploring, selecting and configuring the different game environments.
- **Section 3.3** outlines the selection of the neural network and training platform, including the software architecture used for this project.
- **Section 3.4** presents the selected reinforcement learning algorithms, explaining their principles, implementation details, and preliminary results.
- **Section 3.5** discusses the adaptations, optimisations and techniques implemented to execute the algorithms on embedded devices.
- **Section 3.6** provides a summary of the chapter's contents and introduces the evaluation process presented in the subsequent chapter.

3.1 Design Space Exploration Framework

This project employs a Design Space Exploration (DSE) framework to develop an efficient solution capable of executing DRL algorithms for on-device autonomous game playing while prioritising computational efficiency, stability, real-time adaptability and model accuracy. This section establishes the foundations of the implementation by detailing the proposed methodology, the project requirements, and the design framework used to guide experimentation and optimisation, providing the basis for the subsequent experimentation and optimisation steps.

3.1.1 Overview of the Proposed Methodology

The proposed methodology for developing a DRL model for game playing on embedded systems follows a structured workflow designed to identify and implement the most suitable algorithm for resource-constrained environments. The development process is illustrated in the flowchart represented in Figure 3.1, which outlines the necessary steps to achieve the desired solution.

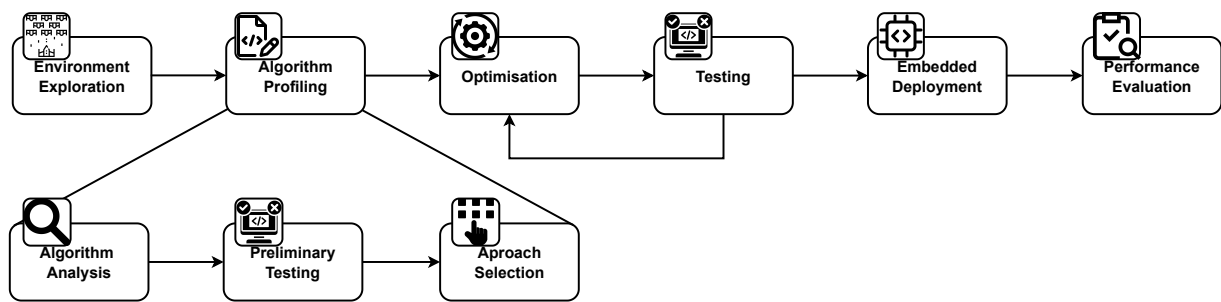


Figure 3.1 Proposed Methodology Flowchart

The first stage of the flowchart corresponds to the environment exploration phase, where different RL environments are explored and compared. The objective of this stage is to select environments that are suitable for both embedded deployment and future evaluation. Factors such as existing implementations, documentation quality, and compatibility with common RL frameworks are taken into account during this phase.

The second stage, algorithm profiling, focuses on analysing candidate algorithms, conducting preliminary tests, and selecting the most suitable approaches for embedded deployment. In this phase, algorithms are considered based on their underlying principles, expected computational demands, performance, and overall popularity. Preliminary experiments will be carried out to gather baseline results for each algorithm, helping identify the most promising approach for further optimisation and evaluation.

After completing the profiling phase, the optimisation phase begins, focusing on refining the selected algorithm to enhance computational efficiency while maintaining accuracy. This includes choosing between different models and implementing optimisation strategies to reduce

memory usage, accelerate inference, and enhance model generalisation. Hyperparameter tuning, network architecture adjustments, and efficient data processing techniques are also considered to further improve performance.

Following the optimisation section, another testing phase will be conducted to validate the improvements made, test the model, and ensure its performance. The main objective of this phase is to identify any remaining instabilities before proceeding to the deployment.

After testing the viability and efficiency of the model, the deployment phase involves transferring it into an embedded system. This process requires adapting the implementation to function within the hardware's computational and memory constraints while maintaining its processing capabilities.

Finally, the performance evaluation phase is conducted to assess the effectiveness of the deployed system. Performance will be measured in terms of learning rates, execution time, accuracy, and resource consumption. Additionally, the trade-offs between computational speed and model performance will be carefully analysed to ensure the system is suited for embedded deployment.

3.1.2 Project Requirements and Target Platform

The main objective of this project is to design and implement a system capable of running DRL algorithms for autonomous game playing on resource-constrained embedded hardware. This requires finding a balance between in-game performance, execution times, and resource usage to ensure that the system achieves reliable performance within its computational limits.

These requirements inevitably introduce trade-offs. Advanced models with complex algorithms often achieve higher scores and demonstrate improved decision-making capabilities. Still, they come at the cost of increased memory usage, longer execution times, and higher energy consumption. Simpler models, on the other hand, run faster and consume fewer resources but may deliver lower performance or require significant amounts of training time to reach comparable results. The work developed in this thesis focuses on addressing these trade-offs by carefully selecting algorithms, applying optimisation techniques, and systematically evaluating their impact on both performance and efficiency, to identify the most suitable algorithms for this task.

For experimentation and validation, the NVIDIA Jetson Orin Nano (Figure 3.2) was chosen as the primary embedded platform. This board integrates a 6-core ARM Cortex-A78AE CPU, a NVIDIA Ampere architecture GPU with 1024 Compute Unified Device Architecture (CUDA) cores and 32 Tensor Cores, 8 GB of memory, and hardware acceleration for AI workloads. It supports configurable power modes ranging from 7 W to 15 W, making it efficient for edge AI applications while still operating under much stricter resource constraints than desktop GPUs. The board provides standard connectivity through multiple USB ports for data transfer and a DisplayPort interface for visual output and user interaction. It is important to emphasise that the proposed approach is not limited to this device but is intended for embedded and low-power systems in general, where similar challenges arise.

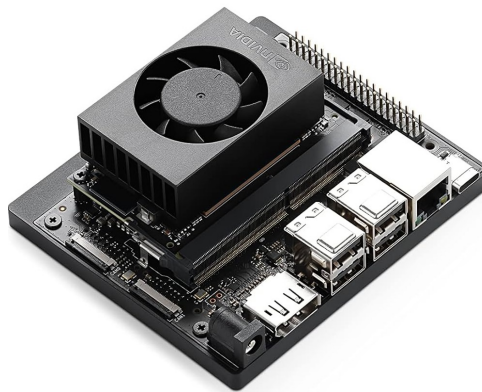


Figure 3.2 NVIDIA Jetson Orin Nano development board [44]

3.1.3 Design Framework

The proposed framework is developed using Python as the primary programming language due to its versatility and strong support for machine learning applications. Its simplicity enables rapid prototyping and seamless integration with various machine-learning frameworks and libraries. Additionally, there is a vast selection of resources and examples created by the community developed with Python, which helps the research phase of this work.

PyTorch serves as the Deep Learning framework, offering dynamic computation graphs and GPU acceleration. Its ease of use makes it particularly suitable for RL tasks, as it enables flexible experimentation with different model architectures and training methodologies.

Gymnasium¹ (formerly known as OpenAI Gym) serves as the main library for RL environments, providing standardised interfaces for various simulation settings. These environments allow for the testing and evaluation of different RL algorithms across a range of games, ensuring consistency in benchmarking and performance assessment.

Additionally, supporting libraries such as NumPy, Pandas, and Matplotlib are used for numerical array operations, dataset handling, and data visualisation, respectively.

NVIDIA's CUDA is also an essential part of the development framework. It provides a computing platform that allows libraries such as PyTorch to offload computationally intensive operations, including convolutions and matrix multiplications, to the GPU, instead of relying on the CPU. Using thousands of cores in parallel, CUDA enables faster training and inference, making it a valuable addition in deep reinforcement learning, where large volumes of data need to be processed efficiently on resource-constrained devices.

¹<https://gymnasium.farama.org/index.html>

3.2 Exploring Game Environments

This section describes the environments and tools used to establish the foundations for training DRL agents. It begins by profiling various environments to identify the most suitable ones for testing, followed by a discussion of the configurations and functionalities applied to them.

3.2.1 Environment Selection

The first stage involved exploring and selecting suitable environments for experimentation. This initial step was essential to identify environments that not only posed appropriate challenges for DRL agents, but also allowed the optimisation and evaluation of different algorithms on embedded hardware, providing a balance between complexity and feasibility.

Various Gymnasium environments were explored to identify the most suitable candidates for experimentation. Classic control environments such as **CartPole** were initially considered, but their simplicity made them unsuitable for testing more advanced DRL techniques, as agents could quickly achieve near-optimal performance. Slightly more complex environments like **Lunar Lander** and **Car Racing** were also tested, but installation issues with the Box2D library made them difficult to set up and unreliable for systematic experimentation. External environments, such as **Flappy Bird**, were also considered, but their sparse reward structure made optimisation harder, and many algorithms have already mastered the game to completion, reducing its usefulness as a challenging benchmark.

Within the Atari suite, several games were also explored, but ultimately not selected. **Q*bert**, while visually appealing, presented highly stochastic gameplay with rewards that were not always well structured, complicating reproducibility. **Pac-Man** offered a rich environment but relied heavily on long-term planning and memory, which would make training much slower for the scope of this work. **Montezuma's Revenge** is a well-known benchmark for sparse reward problems, but its difficulty level is such that even state-of-the-art algorithms struggle to learn effectively, making it unsuitable for the intended comparative experiments.

After this exploration, the focus was narrowed to three Atari games: **Pong**, **Breakout**, and **Space Invaders**. These environments were selected because they offer an incremental scale of complexity, serving as easy, medium and hard tasks, respectively, while remaining feasible for systematic training and optimisation.

Pong (Figure 3.3) is one of the simplest Atari environments and serves as a good starting point for deep reinforcement learning. The agent controls a paddle on one side of the screen and must hit a moving ball to get it past the opponent's paddle. The action space consists of four discrete actions: *noop*, *fire*, *up*, and *down*. The reward structure of this environment is extremely simple, where the agent receives +1 for scoring a point and -1 for conceding one. Since games are played up to a maximum of 21 points, episodes are relatively short compared to other Atari environments, making Pong faster to train. Due to its simplicity, clear reward signals, and well-defined dynamics, Pong was selected as the easy environment for initial testing.



Figure 3.3 Pong Environment Preview [19]

Breakout (Figure 3.4) is a moderately more complex game that serves as a medium level of difficulty. The agent controls a paddle at the bottom of the screen and must bounce a ball upward to break rows of bricks at the top. The action space consists of four discrete actions: *noop*, *fire*, *right*, and *left*. The reward structure varies, depending on what brick the agent hits: breaking a brick in the lowest rows (Blue and Aqua) grants +1, a brick in the middle rows (Green and Yellow) grants +4, and a brick in the highest rows (Orange and Red) grants +7. Although the average human score for this game is around 31 points, the maximum achievable score can go up to 864 points, making episodes significantly longer and providing a richer training environment. Unlike Pong, which always progresses at the same pace, Breakout accelerates as the ball speed increases when higher rows are cleared, introducing an additional layer of difficulty. Furthermore, the game supports multiple modes, allowing the agent's performance to be compared across different variations. Compared to the simpler dynamics of Pong, Breakout requires greater strategic planning, as the agent must keep the ball in play while also aiming trajectories to maximise rewards and efficiently clear the board, making it a decent choice for the medium difficulty environment.

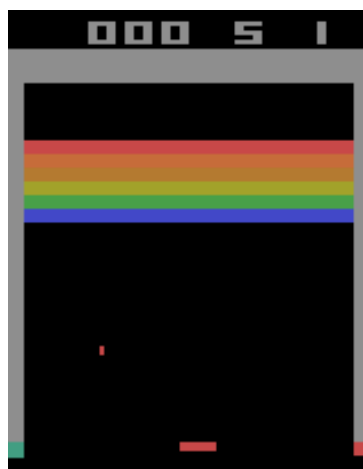


Figure 3.4 Breakout Environment Preview [18]

Space Invaders (Figure 3.5) is the most challenging of the three selected environments. The agent controls a spaceship at the bottom of the screen and must shoot descending waves of aliens while avoiding their attacks. The action space consists of six discrete actions: *noop*, *fire*, *right*, *left*, *rightfire*, and *leftfire*. The reward structure depends on the alien destroyed, with values increasing by row: aliens in the bottom row grant +5 points, with rewards increasing up to +30 for the highest row. In addition, destroying the special ship that occasionally appears yields an extra +200 points. Although the maximum obtainable score for this version of the game is 9,999 points (due to hardware limitations), the average human score tends to be around 1,600 points. The difficulty escalates dynamically, as surviving aliens speed up their movements and attacks the longer they remain in play, forcing the agent to adapt its strategy in real time. Compared to Pong and Breakout, Space Invaders introduces a far greater level of complexity, requiring accurate movement, precise timing for shooting, and the management of multiple simultaneous threats. These characteristics, combined with a larger action pool, make Space Invaders a demanding and highly informative environment to test advanced DRL algorithms, and a good candidate for the hard difficulty level.



Figure 3.5 Space Invaders Environment Preview [20]

3.2.2 Environment Setup and Customisation

Atari environments provide three different observation types that define how agents perceive the game state: RGB, RAM, and grayscale. The RGB observation returns full-colour images of the game screen, offering a detailed representation of the environment but at a higher computational cost, as the agent must process three colour channels at every step. The RAM observation provides the raw internal memory state of the Atari console (128 bytes), and while it is lighter in computational cost, it is harder to train on due to its limited information and lack of spatial structure. Finally, the grayscale observation reduces RGB frames to a single channel, preserving the spatial layout needed for decision-making while lowering memory usage and computation cost. For these reasons, grayscale is generally the most practical choice for training DRL agents in resource-constrained environments.

The environments also support different rendering modes that determine how the game visuals are displayed: `human`, `rgb_array`, and `no render`. The `human` render mode displays the game in real time, allowing for better visualisation of the agent's behaviour. The `rgb_array` mode outputs pixel frames, making it useful for data gathering and analysis tasks. The `no render` mode disables rendering entirely, reducing computational overhead and speeding up training. For most experiments, the `no render` mode is preferred, as it avoids unnecessary processing while maintaining learning efficiency.

Gymnasium also provides the necessary functions to execute environments in synchronous or asynchronous mode. In synchronous mode, training proceeds on a single environment instance with episodes running sequentially, which is more stable but results in longer training times. Asynchronous mode, on the other hand, runs multiple environment instances in parallel, where episodes are executed simultaneously and the acquired knowledge is combined through a shared buffer. This enables faster data collection and reduces training times, but comes at the cost of significantly higher memory usage. During preliminary testing, running multiple environments at the same time often exhausted the available VRAM, making synchronous mode the more reliable choice for resource-constrained hardware.

Gymnasium Wrappers are custom functions that provide an additional layer of flexibility by modifying the way environments behave or how observations and rewards are presented to the agent. They allow the addition of preprocessing, constraints, or adapt the environment without changing its original implementation, which can improve training efficiency.

The Atari Preprocessing Wrapper standardises input observations by resizing frames to a fixed resolution, converting them to grayscale, and applying frame-skipping. These steps reduce computational cost while preserving temporal information, ensuring agents train on simpler and more consistent inputs. All implemented models throughout this project make use of this wrapper to improve learning efficiency and reduce computational cost. Typically, a frameskip of 4 is applied, meaning each action is repeated for four consecutive game frames. This technique reduces the number of unique states and actions the DRL agent must process, which speeds up training and reduces the total computational load while still preserving the essential dynamics of the game environment.

The Frame Stack Wrapper provides the agent with a stack of consecutive frames instead of a single one, allowing it to infer motion and velocity, which is crucial in environments where static images alone are insufficient, such as following the ball in Pong and Breakout or tracking enemy projectiles in Space Invaders.

Reset Wrappers are used to control how environments initialise at the beginning of each episode. They can be used to skip opening sequences, randomise initial states, or automatically execute specific actions required to start the game. These modifications help stabilise training by ensuring that agents begin each episode from a consistent and meaningful state.

For some of the selected environments, the `fire` action is required to start the game, being its only use. To address this, two wrappers were created: `FireResetEnv` and `FireOnLifeLossEnv`, which ensure that the `fire` action is automatically executed either at the start of

the game or after each life loss, respectively. This removes the need for the agent to learn redundant *fire* actions, effectively reducing the action pool and simplifying training. Additionally, a *NoopResetEnv* wrapper was extended, based on similar existing wrappers, to introduce randomness by performing a variable number of initial *noop* actions, improving the agent's generalisation across different starting states.

Reward Wrappers allow tuning the feedback agents receive during training. By reshaping or scaling rewards, they can make learning signals denser, discourage undesirable behaviours, penalise unnecessary actions, or emphasise specific objectives. These wrappers are mainly used to guide agents toward more effective learning strategies and to improve their overall performance, rather than allowing them to exploit unintentional aspects of the reward function.

In Pong, the default reward system is already simple and effective, granting +1 when the agent scores and -1 when the opponent scores. To make evaluation more interpretable, the rewards were split into two categories, *agent_score* and *opponent_score*, allowing a clearer view of how the agent's performance evolves across episodes.

In Breakout, the reward structure was modified to encourage longer and more strategic play. In addition to rewards for breaking blocks, the agent receives small positive rewards for surviving longer without losing a life, and negative rewards each time it fails to hit the ball and loses one of its five lives. To discourage repetitive and ineffective behaviour, a small penalty is also applied if the same action is repeated excessively.

Finally, in Space Invaders, rewards were extended beyond the default system to reinforce survival and discourage poor strategies. Besides points for destroying enemies, the agent receives small positive rewards for staying alive and negative rewards when being hit, losing a life, or losing all three lives. A repetition penalty is also applied to prevent the agent from idling or repeatedly hiding in corners or behind cover.

After selecting the environments, defining the observation settings and applying the necessary wrappers, the groundworks have been established to start the training process. The next step focuses on selecting the most suitable algorithms and applying various optimisation techniques to achieve reliable performance on resource-constrained devices.

3.3 Design and Architecture for Training DRL Algorithms

Designing and optimising a DRL algorithm involves not only the selection of the algorithm itself, but also its neural network architecture, the training platform on which it will train, and all the complementary functions that facilitate the learning process, that is, a structured software architecture to guide its design and exploration.

3.3.1 Neural Network Architectures

Neural networks are a fundamental component of reinforcement learning, as they provide the mechanisms needed to map high-dimensional observations to value estimates, policies, or both. In Atari environments, where each state is composed of thousands of pixels per frame, the

choice of the model architecture directly impacts the agent’s ability to extract relevant features and make decisions. A poorly chosen network design can waste computational resources on redundant information or fail to capture the patterns required for effective gameplay, hindering the learning process.

Since the selected environments are inherently visual, with game states represented as image-like observations, CNNs were chosen as the underlying architecture. Their ability to extract spatial features from pixel data makes them well suited for Atari frames, where identifying patterns such as ball trajectories, paddle positions, or enemy movements is essential.

Architectures such as FNNs, RNNs, transformer networks, and GANs were also considered but found to be less suitable for this task. FNNs lack the spatial inductive bias required for efficient feature extraction from image data. RNNs, while effective for modelling temporal dependencies, are not well-suited to handle high-dimensional pixel observations. Transformer networks, despite having strong representational power, are too expensive for resource-constrained systems. GANs are designed primarily for data synthesis rather than decision-making tasks. Ultimately, the CNNs provides the most effective balance between representational power and computational efficiency, making it the natural choice for Atari-based reinforcement learning.

Defining the activation functions is also a key component of architecture design. ReLU was adopted across all convolutional and fully connected layers because, unlike sigmoid or hyperbolic tangent (*tanh*) activations that suffer from saturation and vanishing gradient issues, they provide a sparse and efficient representation that enables faster training and stable gradient propagation. Softmax is applied only in the output layers of policy networks to generate probability distributions, but not in hidden layers due to its computational cost. Overall, ReLU offers the most effective trade-off between stability, efficiency, and expressiveness for the hidden layers of convolutional architectures in Atari environments.

Based on the standard network architecture for DRL on Atari environments, the CNNs implemented in this project comprises three convolutional layers with kernel sizes of 8×8 , 4×4 , and 3×3 and strides of 4, 2, and 1, respectively, progressively reducing the input resolution while extracting increasingly abstract features. These are followed by a fully connected layer with 512 units, which integrates the extracted features into a compact representation before producing action probabilities (actors), state-value estimates (critics), or both. This parameterisation provides a well-established balance between representational capacity and computational efficiency, with minor adjustments introduced in some algorithms to better match their specific training dynamics.

3.3.2 Training Platforms

The execution of training is highly dependent on the underlying hardware. While CPUs can process neural network operations, they execute tasks sequentially and become significantly slower when handling the large-scale matrix multiplications required by deep learning. GPUs, on the other hand, are designed for parallel operations, allowing thousands of cores to process

data simultaneously. This difference makes GPU acceleration particularly effective for deep reinforcement learning, where large batches of high-dimensional visual data must be processed quickly. For this reason, all algorithms in this project were designed to execute primarily on the GPU (while also offering support for CPU execution).

The performance difference of GPU execution also relies on the use of tensors, which serve as the fundamental data structure in frameworks such as PyTorch. Tensors generalise matrices to higher dimensions and can be mapped directly onto GPU memory. Converting all input data and operations into tensors provided a major advantage for this project, drastically reducing CPU usage by offloading computations to the GPU, resulting in faster and more efficient training.

To fully exploit GPU acceleration on the embedded platform, NVIDIA's CUDA framework provides the necessary programming model that allows libraries such as PyTorch to offload computationally intensive operations like convolutions, matrix multiplications, and gradient updates directly to the GPU. CUDA kernels are further optimised to leverage both CUDA cores and Tensor Cores, significantly reducing runtimes and enabling real-time training.

Configuring CUDA on both the development workstation and the embedded device required several setup steps, such as verifying device compatibility to ensure that the available hardware supported the required CUDA version. This was followed by the installation of NVIDIA's CUDA Toolkit, cuDNN, and other necessary libraries. A CUDA compatible version of PyTorch was installed, allowing tensor operations to run directly on the GPU.

To validate the setup, a simple test script was created (Listing 3.1), which prints the availability of CUDA and identifies the detected GPU. A correct configuration produces an output confirming that PyTorch is successfully using the GPU (Listing 3.2).

```
print(f"CUDA: {torch.cuda.is_available()}")
print(f"GPU: {torch.cuda.get_device_name(0)}")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Listing 3.1 Simple CUDA Device Setup

```
CUDA: True
GPU: Orin
```

Listing 3.2 Correct CUDA Device Setup Output

3.3.3 Utility Functions

To support the execution and evaluation of the explored algorithms, a series of utility functions were implemented. These were designed to simplify experimentation, reduce redundant code, and provide consistent tools that could be reused across different methods.

Model persistence was implemented through save and load functions, which captured the agent’s complete states (including the policy network, target network, and optimiser) and restored it when necessary. This not only allowed training to be paused and resumed, but also supported the transfer of information between models, providing a basis for knowledge distillation.

Plotting functions were developed to better visualise training results. These generate graphs of episode rewards, both as raw scores and with moving averages, enabling a clearer view of learning dynamics. These plots facilitated the interpretation of results and were essential for comparing the effectiveness of different algorithms.

Frame preprocessing utilities were also implemented to convert raw RGB frames into grayscale, crop irrelevant regions, normalise pixel values, and stack consecutive frames into compact state representations. Although these functions offered a lightweight alternative to prepare inputs, they were ultimately discarded in favour of Gym’s built-in wrappers, which provide better optimisation and consistency across all environments.

For memory management, several variants of the *Sum Tree* data structure were implemented, enabling efficient priority-based sampling for replay buffers. By maintaining a binary tree of priorities, these structures achieve logarithmic complexity for both insertions and sampling. Different implementations were explored to handle both standard experiences and tensor-based data, with the addition of mechanisms to prevent memory leaks during training.

Replay buffers formed an essential component of the training pipeline, enabling agents to store experiences and sample them in batches for stable gradient updates. Multiple buffer variants were implemented, ranging from standard replay buffers to asynchronous and memory-optimised versions, ensuring flexibility across different algorithms and hardware setups. Among these, the Prioritized Experience Replay (PER) buffer stood out as one of the most impactful enhancements to agent performance. By assigning higher sampling probabilities to transitions with greater learning potential, PER consistently improved sample efficiency and accelerated convergence. Despite introducing a modest computational overhead, the substantial performance gains justified its inclusion, making PER a beneficial addition to the experimentation setup, even within the limited resources of embedded devices.

3.3.4 Software Architecture

A simplified class diagram (Figure 3.6) was created to illustrate the software architecture used throughout this project. This diagram illustrates the main modules of the system and the relationships between them, showing how they interact to form an efficient reinforcement learning framework. By visualising these interactions, it becomes easier to understand the overall organisation of the code and how different parts contribute to the training process.

The architecture is organised into several main components. The **games** package contains the environments and executable files used for training and evaluation, serving as the foundation of the framework. The **agents** package implements the reinforcement learning algorithms and manages their interaction with the environments. The **models** package defines

the neural network architectures that support the agents. The **wrappers** package includes the custom reset and reward wrappers developed to enhance training. The **utils** package provides supporting functionality such as plotting, saving, and loading models, as well as custom structures like sum trees and replay buffers. Finally, the **trained models** directory stores the results of training, encapsulating the policies learned through interaction with the environments. Together, these components allow algorithms and environments to be combined and tested with minimal modifications to the overall system.

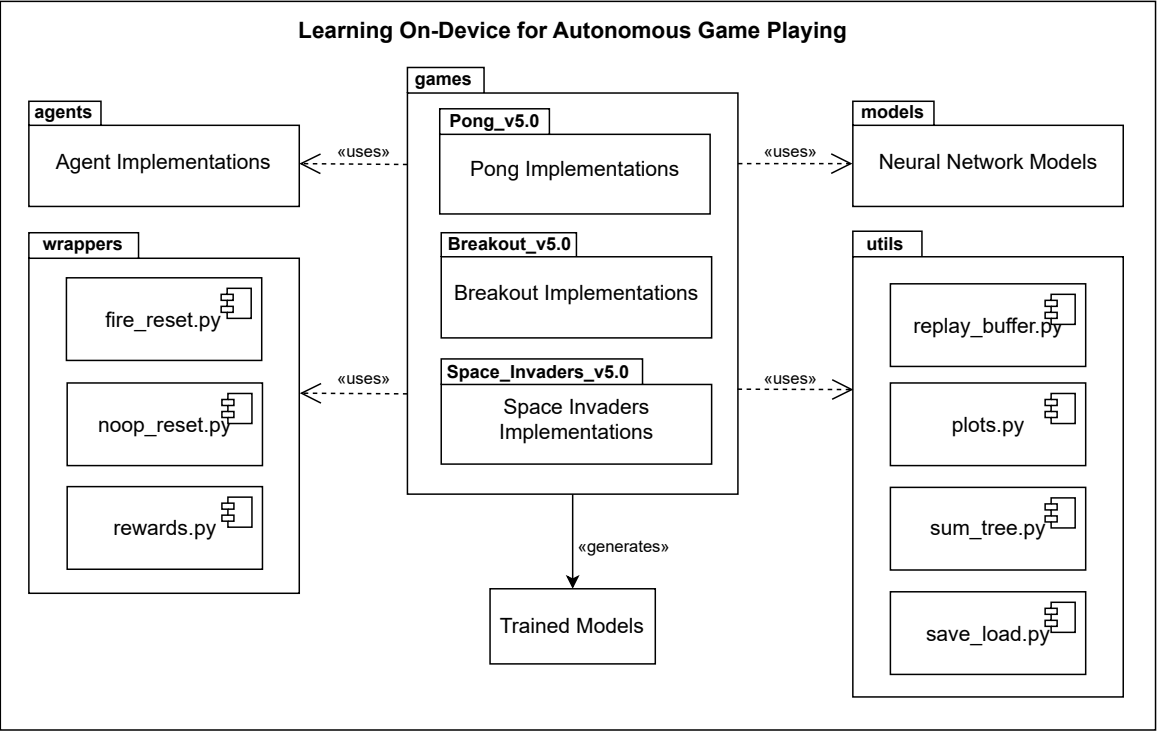


Figure 3.6 Software Architecture Diagram

3.4 Reinforcement Learning Algorithms

Reinforcement learning algorithms form the core of the training process, defining how agents interact with the environment, update their policies, and improve over time. Building on the previously defined CNN, system architecture, and supporting utilities, several algorithms were implemented and tested. In some cases, some modifications were applied, such as adjustments to output layers or the inclusion of stabilisation techniques, which are explained in the descriptions of the corresponding algorithms.

For consistency, all preliminary experiments were carried out in the medium difficulty game environment (*Breakout*), with each method trained for 1000 episodes to ensure a fair comparison. The evaluation focused on their principles, implementation details, preliminary results, and overall suitability for embedded deployment. All experiments were executed on a desktop system equipped with an Intel(R) Core(TM) i5-9600K CPU running at 3.70 GHz, an NVIDIA

GeForce RTX 2070 GPU with 8 GB of VRAM, and 16 GB of system memory (RAM).

The selected algorithms included policy-based, value-based, and actor–critic methods to ensure a variety of strategies were represented, allowing for a broad comparison under the same experimental conditions. This selection consisted of a Random Agent, REINFORCE, TRPO, PPO, A2C, A3C, SAC, DQN, DDQN, DRQv2, and RDQN.

Random Agent

The Random Agent serves as a baseline for evaluating reinforcement learning algorithms. This agent does not learn from interactions with the environment, only selecting actions at random from the available action space, possessing no capacity for improvement or adaptation, but establishing a lower bound of performance against which learning algorithms can be compared. Any algorithm that surpasses this random baseline demonstrates the ability to extract useful patterns from the environment.

Results show that the Random Agent achieved consistently low scores, with slight fluctuations across episodes but no evidence of long-term improvement. The average performance remained minimal, with a high score of 7.0 reached purely by chance (Listing 3.3). The learning curve (Figure 3.7) confirms this behaviour, showing no upward trend in performance, as expected from an agent without learning capability. The script runtime was very short, completing all episodes in just over two minutes due to the absence of training computations.

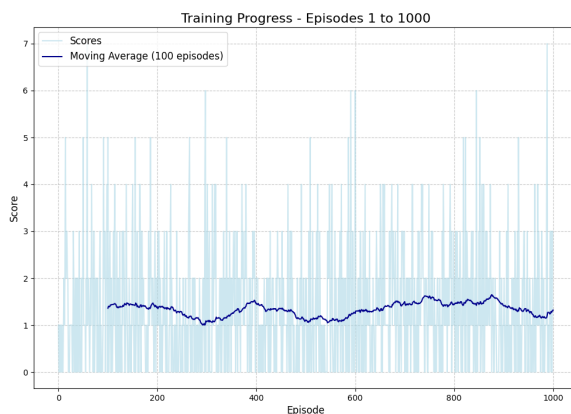


Figure 3.7 Random Agent Learning Curve

Episode 1	Average Score: 1.00
Episode 100	Average Score: 1.36
Episode 200	Average Score: 1.39
Episode 300	Average Score: 1.08
Episode 400	Average Score: 1.45
Episode 500	Average Score: 1.11
Episode 600	Average Score: 1.25
Episode 700	Average Score: 1.50
Episode 800	Average Score: 1.44
Episode 900	Average Score: 1.41
Episode 1000	Average Score: 1.32

Script Runtime: 148 seconds (2.4 min)
Highest Score: 7.0 (Episode 989)

Listing 3.3 Random Agent Scores

From a computational perspective, the Random Agent required very few resources. Memory consumption remained around 0.5 GB, CPU utilisation near 20%, and GPU usage at 0%, since no network computations were involved. These results confirm that while a random strategy is computationally lightweight, its lack of learning ability renders it unsuitable for embedded reinforcement learning.

REINFORCE

The REINFORCE algorithm is one of the first policy-based methods and is often considered a baseline in reinforcement learning. It is a Monte Carlo approach that directly optimises the policy by adjusting its parameters in proportion to the cumulative reward obtained from complete episodes. Actions that yield higher returns become more likely in future decisions, while less successful actions are suppressed. Despite its simplicity, REINFORCE is prone to high variance in gradient estimates, which often leads to unstable training dynamics.

This policy is represented by a convolutional neural network that outputs a probability distribution over actions through a softmax layer. At each step, an action is sampled from this distribution and, once an episode is completed, the policy parameters are updated according to the returned weighted probabilities of the actions taken. This direct optimisation approach avoids the need for a value function but makes the algorithm more sensitive to noisy returns.

For this implementation, the common CNN backbone was maintained with only minor adjustments. A softmax activation in the actor head was used to generate the action probabilities, and the agent was optimised using the Adam optimiser with a learning rate of $LR = 1 \times 10^{-4}$ and a discount factor of $\gamma = 0.99$. Gradient scaling was applied on the GPU to stabilise updates and improve convergence efficiency.

The training process showed only minor progress, with the agent surpassing random behaviour after several episodes, but still obtaining consistently low overall scores. Learning remained unstable, with frequent oscillations in performance between runs. The average scores highlight this variability, with only small improvements in some episodes and a high score that reaches about a third of the average human performance (Listing 3.4). Average scores varied considerably between episodes, with small overall improvement (Listing 3.4). The learning curve further emphasises the inconsistency, where short periods of progress are often followed by regressions (Figure 3.8). Despite its poor learning performance, the execution time was relatively short, standing out as one of the few positive traits of this algorithm.

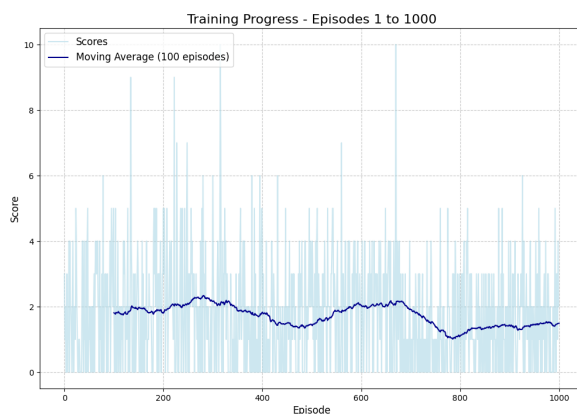


Figure 3.8 REINFORCE Learning Curve

Episode 1	Average Score: 1.00
Episode 100	Average Score: 1.81
Episode 200	Average Score: 1.80
Episode 300	Average Score: 2.16
Episode 400	Average Score: 1.83
Episode 500	Average Score: 1.44
Episode 600	Average Score: 2.04
Episode 700	Average Score: 1.91
Episode 800	Average Score: 1.11
Episode 900	Average Score: 1.43
Episode 1000	Average Score: 1.49

Script Runtime: 996 seconds (16.6 min)
Highest Score: 10.0 (Episode 671)

Listing 3.4 Average REINFORCE Scores

From a computational perspective, REINFORCE remained lightweight throughout execu-

tion. Memory usage averaged around 2.5 GB, CPU utilisation stayed between 20–25%, and GPU load was consistently in the 90–95% range. These values show that the algorithm makes efficient use of available hardware without overloading system resources.

Despite this efficiency, REINFORCE does not appear promising for embedded deployment. Its limited learning progress, unstable training behaviour, and consistently low performance outweigh the advantages of fast execution and modest resource usage. More advanced methods, which reduce variance by incorporating value function estimation, provide a stronger and more reliable basis for efficient embedded reinforcement learning.

Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization (TRPO) is a policy-based method that builds on the principles of REINFORCE by introducing a trust-region constraint to stabilise learning. Instead of relying on unconstrained gradient ascent, TRPO restricts the change between successive policies using a KL divergence bound. KL divergence is a measure of how one probability distribution diverges from another, and in this context it ensures that the updated policy remains close to the previous one. This mechanism prevents excessively large updates, reducing the risk of catastrophic drops in performance while still enabling efficient exploration of the policy space.

The implementation used the previously defined CNN for feature extraction. Policy updates were performed using the conjugate gradient method with ten iterations and a line search of up to ten steps, with a KL divergence constraint of 0.01 to enforce stability. Generalised advantage estimation (GAE, $\lambda = 0.95$) was applied to compute low-variance advantages, and an entropy bonus was included to encourage exploration. Optimisation was also done using the Adam algorithm with learning rates of 2.5×10^{-4} , and updates were performed every 2048 steps to balance computational efficiency with policy improvement.

Training results showed that TRPO achieved average scores that initially improved but soon stagnated at low values. The agent demonstrated some ability to surpass random behaviour, but overall performance remained unstable, with frequent fluctuations across episodes. The highest score of 9.0 was reached multiple times but never exceeded, leaving average scores with little improvement across the 1000 training episodes (Listing 3.5). The learning curve further highlights this instability, as initial progress was followed by long periods of stagnation (Figure 3.9). The execution time was considerably longer than that of similar lighter models, due to the heavy computational requirements of the conjugate gradient and line search procedures. Each update required several conjugate gradient iterations, repeated line searches, and careful monitoring of the KL divergence constraint, resulting in a slower overall training.

From a computational perspective, TRPO consumed around 1.5 GB of memory, maintained CPU utilisation between 25–30%, and sustained GPU load of 85–90%. These values indicate that the algorithm made efficient use of resources, but its update mechanism made training much more time-consuming than expected.

The training results suggest that the TRPO algorithm, despite its theoretical advantages, does not provide enough performance gains to justify its excessive runtime. Its instability

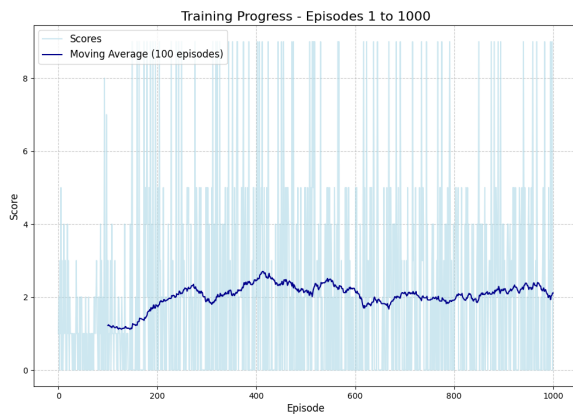


Figure 3.9 TRPO Learning Curve

Episode 1	Average Score: 1.00
Episode 100	Average Score: 1.22
Episode 200	Average Score: 1.76
Episode 300	Average Score: 1.90
Episode 400	Average Score: 2.46
Episode 500	Average Score: 2.17
Episode 600	Average Score: 2.17
Episode 700	Average Score: 2.13
Episode 800	Average Score: 1.92
Episode 900	Average Score: 2.18
Episode 1000	Average Score: 1.11

Script Runtime: 2708 seconds (45.4 min)
Highest Score: 9.0 (Episode 150)

Listing 3.5 Average TRPO Scores

and persistently low scores also restrict its suitability for embedded deployment, especially when other policy-based alternatives offer better stability and performance with faster execution times.

Proximal Policy Optimisation (PPO)

Proximal Policy Optimisation (PPO) is a policy-gradient method designed to stabilise training while preserving sample efficiency. Although classified as policy-based, it is commonly implemented in an actor–critic framework, where the actor directly parameterises the policy and the critic estimates state values to reduce gradient variance. Its main advantage is the clipped surrogate objective, which constrains policy updates by limiting the change in action probabilities between consecutive policies. This prevents destructive updates, leading to steadier learning and making PPO one of the most adopted algorithms for high-dimensional control tasks.

The implementation followed the established CNN for visual feature extraction, extended with separate actor and critic heads. The actor outputs a categorical distribution over actions, while the critic estimates state values that serve as a baseline for variance reduction. A Generalised advantage estimation (GAE $\lambda = 0.95$) was used to compute low-variance advantages. Policy updates relied on the clipped objective with $\epsilon = 0.2$, complemented by an entropy bonus to encourage exploration, while the critic was optimised with mean squared error. Training employed the Adam optimiser with learning rates of $LR = 2.5 \times 10^{-4}$ for both the actor and critic, a discount factor $\gamma = 0.99$, batch size 128, ten epochs per update, and rollouts of 2048 steps between updates. Gradient clipping and mixed-precision scaling on the GPU were also enabled to improve numerical stability and throughput.

The training process produced more convincing results than simpler policy-based methods, with scores showing steady upward progress. The average performance improved consistently across episodes, even reaching a high score of 26.0, which is close to the average human performance (Listing 3.6). The learning curve shows smoother improvements and fewer regressions, aligning with this algorithm’s design to limit destructive updates (Figure 3.10). The

execution time was longer due to multiple epochs per update and critic optimisation, but it remained manageable for experimentation.

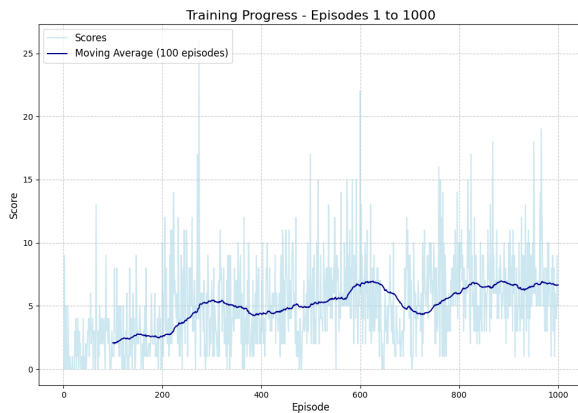


Figure 3.10 PPO Learning Curve

Episode 1	Average Score: 0.00
Episode 100	Average Score: 2.07
Episode 200	Average Score: 2.55
Episode 300	Average Score: 5.44
Episode 400	Average Score: 4.37
Episode 500	Average Score: 5.12
Episode 600	Average Score: 6.55
Episode 700	Average Score: 4.93
Episode 800	Average Score: 5.99
Episode 900	Average Score: 6.75
Episode 1000	Average Score: 6.65

Script Runtime: 1927 seconds (32.1 min)
Highest Score: 26.0 (Episode 275)

Listing 3.6 Average PPO Scores

From a computational perspective, PPO demonstrated moderate resource usage during training. Memory usage averaged around 1.5 GB, CPU utilisation remained stable between 20–25%, and GPU load was consistently around 90%. These values reflect the additional cost of mini-batch updates and multiple training epochs but remain manageable for both desktop and embedded hardware.

Overall, PPO proved to be far more effective than simpler policy-based methods, due to the clipped objective allowing steady improvements in performance while maintaining stable training dynamics. The algorithm’s relatively small computational cost, combined with its consistent learning progress, made it a promising candidate for embedded deployment. Although execution times were longer than in lighter approaches, the improved performance justified the additional cost.

Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C)

The Advantage Actor-Critic (A2C) algorithm builds upon policy-gradient methods by introducing a value function baseline to reduce the variance of policy updates. Instead of relying solely on full returns, it uses the critic’s state-value estimates to stabilise the training signal. The actor adjusts the policy to increase the likelihood of advantageous actions, while the critic evaluates states to guide these updates more effectively. This combination makes A2C one of the simplest yet most effective actor–critic frameworks.

The Asynchronous Advantage Actor-Critic (A3C) extends A2C by running multiple agents in parallel environments, each updating a shared global network, leading to a faster training process. However, implementing this architecture required major modifications to the code structure, particularly to manage worker synchronisation and handle multiple environments simultaneously. Given these complexities, and since the main change lies in asynchronous execution rather than the learning mechanism itself, the performance of A3C will be represented

by the A2C implementation.

For this implementation, the standard CNN was maintained, with two separate heads: the actor produced action logits (the raw, unnormalized outputs of a neural network) via a softmax distribution, and the critic estimated state values. Both networks were trained using the Adam optimiser with learning rates of 2.5×10^{-4} , alongside a discount factor of $\gamma = 0.99$, a value loss coefficient of 0.5, and an entropy coefficient of 0.05 to encourage exploration. Training updates occurred every 256 steps, and the total loss combined actor, critic, and entropy terms, with entropy regularisation preventing premature convergence to suboptimal policies.

The training process revealed limited improvements, with the agent only marginally surpassing random behaviour across 1000 episodes. Scores oscillated throughout runs without showing a consistent upward trajectory. The recorded results show minor gains with a highest score of 10.0 (Listing 3.7), while the learning curve highlights unstable learning dynamics with no clear long-term progress (Figure 3.11). Despite this weak performance, runtime was relatively short, which stands out as one of the algorithm's advantages.

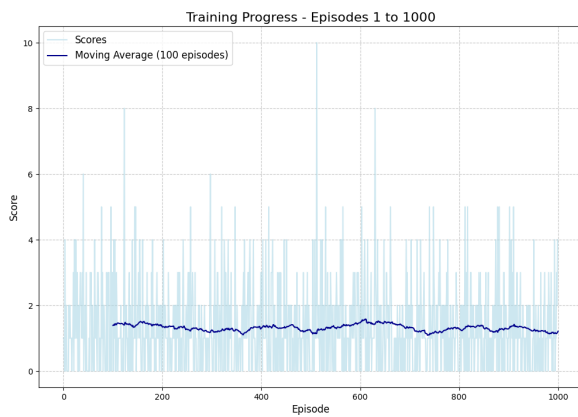


Figure 3.11 A2C Learning Curve

Episode 1	Average Score: 1.00
Episode 100	Average Score: 1.39
Episode 200	Average Score: 1.34
Episode 300	Average Score: 1.25
Episode 400	Average Score: 1.30
Episode 500	Average Score: 1.22
Episode 600	Average Score: 1.47
Episode 700	Average Score: 1.34
Episode 800	Average Score: 1.28
Episode 900	Average Score: 1.29
Episode 1000	Average Score: 1.21

Script Runtime: 629 seconds (10.4 min)
Highest Score: 10.0 (Episode 513)

Listing 3.7 Average A2C Scores

From a computational perspective, the A2C algorithm was lightweight, with memory usage averaging around 2 GB, CPU utilisation between 25–30%, and GPU load in the 85–90% range. These values confirm its efficiency in making use of available hardware resources.

Despite its computational efficiency and short runtime, the weak learning performance and lack of stability suggest that A2C is not a promising candidate for embedded deployment. More advanced actor–critic methods with additional stabilisation mechanisms are required to achieve reliable performance.

Soft Actor-Critic (SAC)

The SAC algorithm is an off-policy actor-critic method that extends standard reinforcement learning with the principle of maximum entropy. Instead of optimising for cumulative reward, SAC introduces an additional entropy term in the objective, encouraging the policy to remain stochastic. This promotes more exploration by discouraging premature convergence to subop-

timal deterministic strategies. This method can also be adapted to discrete domains, where the policy outputs a categorical distribution over actions.

The implementation used the common CNN for visual feature extraction, followed by separate actor and critic networks. The actor produced logits that parameterised a categorical distribution over actions, while two independent Q-networks were trained to mitigate overestimation bias. Each Q-network was paired with a target network, updated via Polyak averaging ($\tau = 0.005$), a smoothing technique that blends a small portion of the current parameters into the target networks at each step to ensure stable learning. Replay memory was introduced through a buffer with a capacity of 50,000 transitions, from which batches of 32 samples were drawn. Both the actor and critics were trained with the Adam optimiser at a learning rate of 5×10^{-4} , and the entropy coefficient ($\alpha = 0.2$) was automatically adjusted during training to balance exploration and exploitation.

Training results showed small progress, with the agent consistently surpassing random behaviour but failing to achieve reliable scores. The average scores improved gradually in the early episodes, peaking around 2.0, before stagnating and fluctuating without a clear long-term improvement. The recorded results (Listing 3.8) reflect this limited learning, with small early gains and a high score of 9.0 reached around a quarter of the way in the training process, but never exceeded afterward. The learning curve confirms this behaviour, showing an initial phase of improvement followed by oscillations and plateauing (Figure 3.12). The execution time was notably long, with training taking almost two hours to complete. This overhead originated from the use of twin Q-networks, replay buffer sampling, and entropy-regularised updates, all of which increased the number of forward and backward passes required at each step.

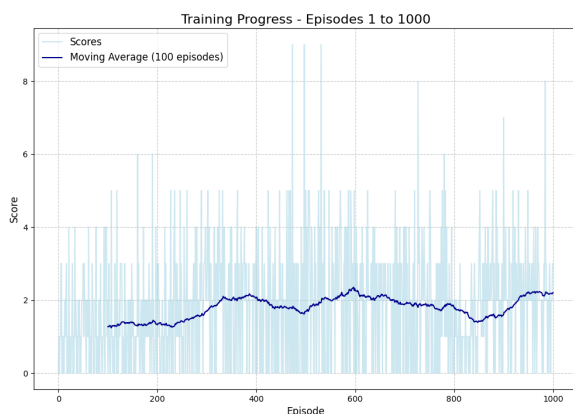


Figure 3.12 SAC Learning Curve

```
Episode 1 Average Score: 0.00
Episode 100 Average Score: 1.27
Episode 200 Average Score: 1.32
Episode 300 Average Score: 1.71
Episode 400 Average Score: 2.06
Episode 500 Average Score: 1.69
Episode 600 Average Score: 2.27
Episode 700 Average Score: 1.89
Episode 800 Average Score: 1.80
Episode 900 Average Score: 1.60
Episode 1000 Average Score: 2.20
```

```
Script Runtime: 3927 seconds (65.4 min)
Highest Score: 9.0 (Episode 474)
```

Listing 3.8 Average SAC Scores

From a computational perspective, SAC required substantially more resources compared to previous methods. Memory usage ranged from 5–6 GB, largely due to the replay buffer, while CPU utilisation remained around 25–30% and GPU load reached 95%. These values demonstrate the heavy computational footprint of this algorithm.

The combination of poor performance, long training times, and high resource usage makes

SAC poorly suited for embedded devices. Although its entropy-regularised formulation aims to encourage exploration, its results do not justify the substantial computational overhead. Consequently, SAC does not appear to be a promising candidate for embedded deployment under these conditions.

Deep Q-Network (DQN)

The DQN algorithm is a value-based method which, unlike policy-based approaches, directly learns a state-action value function (Q -function), estimating the expected return of taking an action a in a given state s . The policy is derived by selecting the action with the highest predicted Q -value, while exploration is ensured through an ϵ -greedy strategy. The addition of experience replay breaks the correlation between consecutive samples by randomly drawing batches from a replay buffer, making the gradient signal more stable and efficient.

The training process minimises the TD error between the predicted and target values, using the standard update rule formula (Equation 3.1), where θ represents the parameters of the current network, θ^- the parameters of the target network, and γ the discount factor. The target is the immediate reward plus the discounted maximum future value, while the loss penalises the difference between the predicted and target values. The target network is maintained and updated through soft updates, reducing the risk of oscillations and further stabilising training.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_{\theta}(s, a) \right)^2 \right] \quad (3.1)$$

The implementation used the standard CNN, followed by fully connected layers to predict Q -values for each possible action. Experience replay was implemented with a buffer of 100,000 transitions, from which batches of 64 samples were drawn for updates. The target network was updated using soft updates with $\tau = 0.001$, and training employed the Adam optimiser with a learning rate of 2.5×10^{-4} . An ϵ -greedy exploration strategy was utilised, with ϵ decaying exponentially over time, while network updates were performed every four environment steps. Long-term rewards were incorporated using a discount factor of $\gamma = 0.99$, balancing short-term gains with future outcomes.

The training results showed steady improvements after an initial slow start, with average scores beginning near random levels, but improving consistently halfway through the training process (Listing 3.9). The agent demonstrated the capacity to exceed the average human performance, achieving a maximum score of 38.0 towards the final episodes. The learning curve (Figure 3.13) highlights this trajectory, showing a clear acceleration in performance after the midpoint of training and only moderate fluctuations near the end. These results demonstrate the effectiveness of DQN in discrete control tasks, where its value-based approach takes advantage of the structure of the environment to achieve strong results.

From a computational standpoint, DQN presented moderate resource demands. CPU utilisation remained between 25–30%, and GPU load was consistently between 90 and 95%. Memory usage was notably higher at 5–6 GB, primarily due to the large replay buffer required by value-based methods. The script runtime was roughly double that of most policy-based

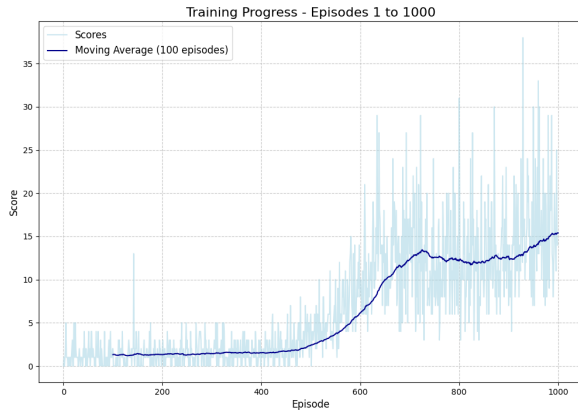


Figure 3.13 DQN Learning Curve

```

Episode 1 Average Score: 0.00
Episode 100 Average Score: 1.32
Episode 200 Average Score: 1.31
Episode 300 Average Score: 1.45
Episode 400 Average Score: 1.53
Episode 500 Average Score: 2.41
Episode 600 Average Score: 6.14
Episode 700 Average Score: 12.34
Episode 800 Average Score: 12.17
Episode 900 Average Score: 12.68
Episode 1000 Average Score: 15.43

```

```

Script Runtime: 1697 seconds (28.2 minutes)
Highest Score: 38.0 (Episode 930)

```

Listing 3.9 Average DQN Scores

methods, but the gains in stability and performance provided a balanced trade-off between computational requirements and learning performance.

These results suggest that DQN is a strong candidate for embedded deployment, since it provided stable learning dynamics and achieved above average scores. The large memory usage may seem like a potential limitation, but can be mitigated by reducing the experience replay buffer size, allowing the algorithm to adapt more effectively to resource-constrained environments without compromising performance.

Double Deep Q-Network (DDQN)

The DDQN algorithm expands on the original DQN by addressing the issue of overestimation bias in action-value predictions. Rather than using the same network to both select and evaluate the next action, DDQN uses the online network to select the action and the target network to evaluate its value. This separation reduces the upward bias introduced by taking a max over noisy estimates and leads to more stable value estimates and improved training stability.

The training process minimises the TD error between the predicted and target values, but unlike DQN, the target is modified to avoid overestimation (Equation 3.2). The online network Q_θ is used to choose the action, while the target network Q_{θ^-} evaluates it, with γ representing the discount factor. This separation prevents the maximisation step from being applied twice to the same estimates, reducing bias and improving stability.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma Q_{\theta^-}(s', \arg \max_{a'} Q_\theta(s', a')) - Q_\theta(s, a) \right)^2 \right] \quad (3.2)$$

The implementation retained the same convolutional backbone used in DQN for visual feature extraction, but replaced the fully connected head with a duelling network architecture. In this design, the Q -value is decomposed into a state-value stream $V(s)$ and an advantage stream $A(s, a)$, which are later recombined into the final estimate. This separation allows the network to better recognise the importance of states, even when individual actions have little immediate effect. A PER mechanism was employed, where transitions are sampled with prob-

abilities proportional to their TD-error, increasing the frequency of updates on more informative experiences.

Training was carried out using the Adam optimiser with a learning rate of 2.5×10^{-4} , a PER buffer of 100,000 transitions, and batches of 64 samples. An ϵ -greedy exploration strategy was applied, decaying ϵ exponentially throughout the training process. Network updates were performed every four environment steps, while the target network was updated through soft updates with $\tau = 0.001$, and a discount factor of $\gamma = 0.99$ to balance immediate and long-term rewards.

Training results started at random levels but, gradually improved as learning progressed (Listing 3.10). The average scores continuously increased, reaching values of around 20 points by the end of training and a maximum score of 50.0, nearly double the average human performance. The learning curve (Figure 3.14) demonstrates this behaviour, showing rapid improvements halfway through training, followed by fluctuations around moderately high scores. The frequent peaks of strong performance, highlight the algorithm’s capacity to discover effective policies.

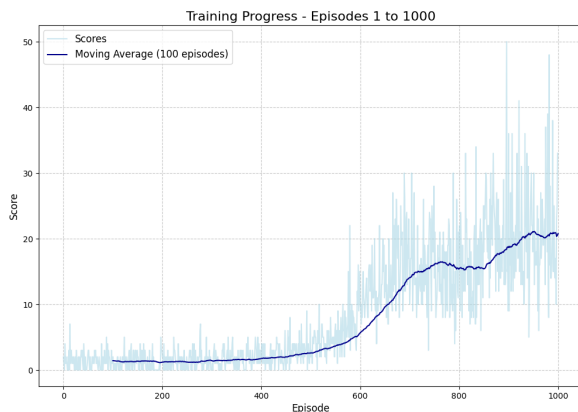


Figure 3.14 DDQN Learning Curve

```

Episode 1 Average Score: 0.00
Episode 100 Average Score: 1.42
Episode 200 Average Score: 1.20
Episode 300 Average Score: 1.41
Episode 400 Average Score: 1.74
Episode 500 Average Score: 2.61
Episode 600 Average Score: 5.67
Episode 700 Average Score: 14.07
Episode 800 Average Score: 15.41
Episode 900 Average Score: 18.65
Episode 1000 Average Score: 20.77

```

```

Script Runtime: 2139 seconds (35.6 minutes)
Highest Score: 50.0 (Episode 897)

```

Listing 3.10 Average DDQN Scores

From a computational perspective, DDQN required a heavier resource usage compared to its predecessor. The CPU and GPU utilisation remained the usual 25–30% and 90–95% respectively, but memory usage went up to 6–7 GB, primarily due to the PER buffer. Despite an execution time roughly double that of most policy-based algorithms and slightly above the value-based ones, the algorithm’s stronger performance makes it a reasonable trade-off.

These results suggest that DDQN is a very solid candidate for embedded deployment, given its ability to achieve stable learning and high scores. The elevated memory usage caused by the large replay buffer is the main limiting factor, but this can also be alleviated by reducing buffer size at the cost of a slightly slower convergence, allowing adaptation to resource-constrained environments while preserving the algorithm’s performance.

Data-Regularized Q-learning (DRQv2)

The Data-Regularized Q-learning (DRQv2)[65] algorithm is an improvement over the common Q-learning by integrating data augmentation and double Q-learning with clipped targets. DRQv2 applies randomised transformations such as brightness and noise perturbations during training, to improve the generalisation to unseen states. The algorithm maintains two Q-networks and two target networks, and the minimum of their predictions is used to compute the loss. This clipped double Q-learning approach reduces overestimation bias while leveraging augmentation to stabilise the training of value functions from pixel inputs.

The implementation, based on the algorithm’s author own implementation², used a convolutional encoder followed by a projection layer and two independent Q-heads (Q_1, Q_2) to estimate action values. Data augmentation was applied during training to diversify state representations, and a replay buffer of 100,000 transitions was used to enhance updates. The agent was trained with the Adam optimiser at a learning rate of 1×10^{-4} , batches of 128 samples, and a discount factor of $\gamma = 0.99$. The target network was updated through soft updates with $\tau = 0.01$, while exploration was handled through a noise-based policy whose standard deviation decayed progressively during training.

Training results were initially poor but showed gradual and steady improvement as experience accumulated (Listing 3.11). The scores climbed consistently without substantial oscillations, reaching average values of around 13.5 points and a high score of 33.0 points, just surpassing the human average. The learning curve (Figure 3.15) shows this progression, with slow but reliable gains over time, highlighting the role of augmentation, although at the cost of slower convergence compared to other value-based methods.

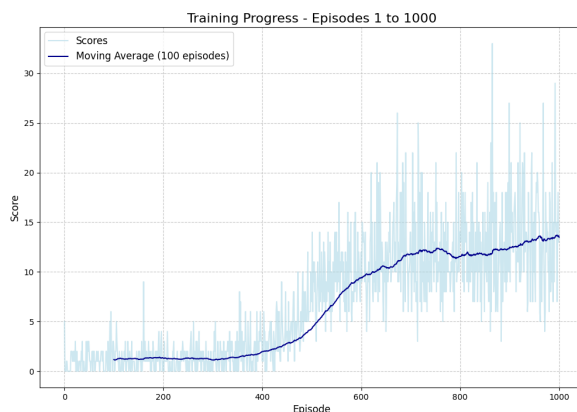


Figure 3.15 DRQv2 Learning Curve

```
Episode 1 Average Score: 0.00
Episode 100 Average Score: 1.16
Episode 200 Average Score: 1.31
Episode 300 Average Score: 1.13
Episode 400 Average Score: 1.97
Episode 500 Average Score: 4.29
Episode 600 Average Score: 9.40
Episode 700 Average Score: 11.81
Episode 800 Average Score: 11.55
Episode 900 Average Score: 12.47
Episode 1000 Average Score: 13.50
```

```
Script Runtime: 3805 seconds (63.4 minutes)
Highest Score: 33.0 (Episode 866)
```

Listing 3.11 Average DRQv2 Scores

From a computational perspective, the DRQv2 algorithm took around an hour to complete, with CPU utilisation averaging around 30%, GPU load maintained at 90–95%, and memory usage at approximately 2 GB. Although its resource demands were relatively modest, the extended runtime derives from the use of dual Q-networks, which effectively double the workload

²<https://github.com/facebookresearch/drqv2>

during both forward and backward passes.

These results suggest that DRQv2 has potential for embedded deployment, as its resource usage is modest compared to other deep RL methods and its performance is generally reasonable. However, its main weakness lies in the long training time, which reduces its practicality in scenarios where speed is preferred over performance.

Rainbow Deep Q-Network (RDQN)

The Rainbow Deep Q-Network (RDQN) algorithm, as previously mentioned, extends the classic DQN by integrating several techniques into a single framework. It incorporates DDQN to mitigate overestimation bias, Duelling Networks to separate state-value and advantage estimation, Distributional DQNs (C51) to represent the full distribution of returns, and Noisy Networks to replace the ϵ -greedy exploration with parameterised noise in the network weights. In addition, it employs PER to increase sampling efficiency and multi-step returns to speed up reward propagation, resulting in a more sample-efficient value-based learning algorithm.

Training aims to minimise the TD-error, by combining Double Q-learning with a distributional setting (Equation 3.3). The policy network selects the next action, the target network evaluates it, and the categorical distribution over returns is projected onto a fixed support $[V_{\min}, V_{\max}]$. The KL divergence between predicted and target distributions defines the loss, aligning the learned return distribution with the true return distribution, leading to a reduced overestimation.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')} \left[D_{\text{KL}} \left(\Phi \left(r + \gamma^n Q_{\theta^-} \left(s', \arg \max_{a'} Q_{\theta} (s', a') \right) \right) \parallel Q_{\theta} (s, a) \right) \right] \quad (3.3)$$

The implementation used a convolutional encoder followed by duelling streams, with Noisy Linear layers replacing standard fully connected layers to inject exploration through parameterised noise. The categorical distribution was represented with 51 atoms (C51 head) spanning between $[-10, 10]$. Experience replay was managed by a PER buffer of 100,000 transitions, and multi-step bootstrapping with $n = 3$ was applied. Training used the Adam optimiser with a learning rate of 6.5×10^{-5} , with batches of 128 samples and a discount factor of $\gamma = 0.99$.

The training results started at random levels but showed rapid convergence, with the agent reaching decent scores much earlier than most algorithms (Listing 3.12). The average scores increased steadily, reaching values of around 10.0 in the later episodes, while the agent achieved a maximum score of 28.0, closing in on the average human performance. The learning curve (Figure 3.16) illustrates this behaviour, showing a quick initial improvement with a consistent upward trajectory that reflects the algorithm's stability and strong sample efficiency. Rainbow is usually intended for very long training sessions, so its performance at this stage, while slightly lower than expected, is not a limitation of the algorithm itself, but a consequence of the reduced training periods used for preliminary testing.

From a computational perspective, Rainbow required an excessive use of resources, with CPU and GPU loads consistently above 95% and memory usage around 8 GB, caused by the combination of PER and categorical outputs. The execution time was extremely long, when

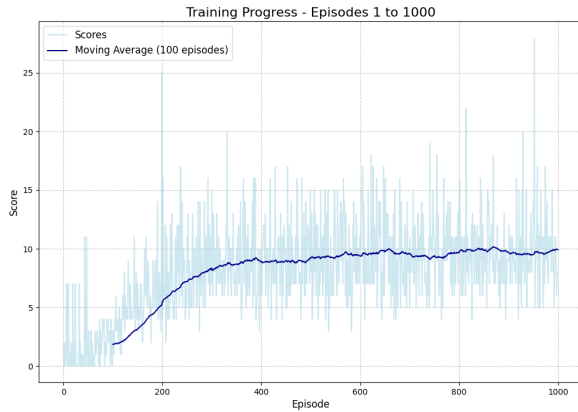


Figure 3.16 RDQN Learning Curve

```

Episode 1 Average Score: 0.00
Episode 100 Average Score: 1.84
Episode 200 Average Score: 5.49
Episode 300 Average Score: 8.16
Episode 400 Average Score: 8.92
Episode 500 Average Score: 9.22
Episode 600 Average Score: 9.38
Episode 700 Average Score: 9.58
Episode 800 Average Score: 9.70
Episode 900 Average Score: 9.69
Episode 1000 Average Score: 9.92

```

```

Script Runtime: 12145 seconds (202.4 minutes)
Highest Score: 28.0 (Episode 953)

```

Listing 3.12 Average RDQN Scores

compared to other value-based methods, mainly due to the added complexity of distributional targets and multi-step updates being maintained simultaneously.

These results confirm that RDQN is a powerful but computationally heavy algorithm. Its strong performance and stability make it a decent choice for scenarios where resources are plentiful, but the increased runtime and computational cost reduce its suitability for resource-limited embedded systems. In such cases, simpler value-based methods may offer a more balanced trade-off between efficiency and performance.

3.4.1 Comparison of DRL Algorithms for Game Playing

To analyse the results obtained from the preliminary testing, a comparison of all algorithms tested was summarised in a table (Table 3.1). This table gathers some of the performance indicator metrics, to help identify trade-offs between learning effectiveness, runtime, and resource consumption, while also highlighting the strengths and limitations of each method.

Algorithm	Best Avg. Score	High Score	Runtime (min)	CPU (%)	GPU (%)	RAM (GB)
REINFORCE	2.04	10.0	16.6	25	95	2.5
TRPO	2.46	9.0	45.4	30	90	1.5
PPO	6.75	26.0	32.1	25	90	1.5
A2C / A3C	1.47	10.0	10.4	30	90	2
SAC	2.27	9.0	65.4	30	95	6
DQN	15.43	38.0	28.2	30	95	6
DDQN	20.77	50.0	35.6	30	95	7
DRQv2	13.50	33.0	63.4	30	95	2
RDQN	9.92	28.0	202.4	95	95	8

Table 3.1 Tested Algorithms Comparison

From the results, clear distinctions can be observed between the algorithm families. Value-based methods consistently achieved the highest performances, while policy-based methods

tended to yield lower results but generally demanded fewer computational resources. Actor–critic approaches did not particularly stand out, with results failing to demonstrate any significant advantage over the lighter policy-based methods or the stronger value-based algorithms.

Among the policy-based methods, PPO emerged as the most reliable algorithm. It achieved moderate scores, got close to average human performance on its highest score, maintained stable learning, and kept resource consumption relatively low. In comparison, REINFORCE and TRPO struggled to achieve meaningful improvements, with training results remaining close to random behaviour with the same amount of training.

For the value-based approaches, DQN and DDQN delivered the best overall results, combining decent scores with high, but adaptable, resource usage. The DRQv2 algorithm also proved promising, achieving performance close to the stronger algorithms while maintaining lower memory demands. However, its extremely long runtime represents a major drawback, limiting its practicality.

The RDQN stands out as a state-of-the-art method that integrates multiple advanced techniques into a single framework. While it demonstrated a fast and stable convergence and good learning efficiency, these benefits came at the cost of very high computational requirements and excessive runtime. This makes it less suitable for constrained environments despite its appeal.

By comparing the different algorithms, a clear trade-off between performance and efficiency was revealed. Value-based methods consistently delivered the strongest results but at a higher computational cost, while policy-based methods proved to be more resource efficient, but achieved lower performances. Actor–critic approaches occupied a middle ground, failing to offer a decisive advantage that would justify their complexity. These comparisons provided valuable information for selecting the appropriate algorithms for deployment in embedded devices.

3.5 Optimising DRL Algorithms for Embedded Computing

The deployment of reinforcement learning algorithms on embedded hardware introduces unique challenges compared to desktop environments, such as less computational power and limited resources, requiring additional preparations and optimisations to be applied. After analysing and comparing the performance of all candidate algorithms, the most promising ones were selected for deployment on the embedded device, with some additional setup, code adaptations, and adjustments being necessary to ensure they could operate effectively under these constraints.

Setting up the NVIDIA Jetson Orin Nano involved configuring both the hardware and the software environment. This process included installing the Linux based operating system with the appropriate interface, enabling CUDA support for GPU acceleration, and verifying that the device drivers were properly configured. The required programming language and libraries

were also installed, along with any additional dependencies needed to run the RL agent's code on the embedded system.

The selection of algorithms for embedded deployment depended on multiple evaluation metrics. These included average and maximum scores achieved, training stability, resource usage, and overall script runtime. By considering these factors, it was possible to identify the algorithms that offered a good compromise between performance and computational feasibility. The algorithms that seemed most promising for embedded deployment, due to their strong performances, efficient use of resources, or a combination of both, were the PPO, the DQN, and the DDQN. DRQv2 and RDQN also appeared to be strong candidates, with the former achieving good scores at modest resource usage but requiring excessively long runtimes, and the latter showing fast convergence but at the cost of being highly computationally demanding.

The first major optimisation strategy implemented was to change most data structures to tensors, ensuring that the bulk of computation was shifted from the CPU to the GPU. This reduced the load on the processor and allowed more efficient use of the Jetson's hardware accelerators.

Mixed-precision computation (FP16) was also employed, where tensor operations are executed in half-precision format. This modification reduces the computational cost of the models and accelerates training and inference, particularly when using GPUs with dedicated FP16 hardware support. By lowering numerical precision without severely impacting accuracy, FP16 enables larger batch sizes and faster computation, improving efficiency on resource-constrained platforms.

Knowledge distillation was another considered technique, as it provides an alternative to training directly from a cold start on the embedded system. In a cold start scenario, the agent begins learning entirely from scratch, requiring longer execution times. Knowledge distillation mitigates this by first training a model on a more powerful device, and then transferring its knowledge to the weaker device. This allows the embedded agent to start from an informed state, leveraging prior experience while still retaining the ability to refine its behaviour through additional interaction with the environment. However, this technique was not utilised in the primary experiments of this thesis, as the main focus was to demonstrate the feasibility of full on-device training from scratch. Knowledge distillation would only be considered as a fallback solution if training directly on the embedded device proved infeasible.

These adaptations and optimisation strategies ensured that the selected algorithms could be deployed effectively within the hardware constraints. With these preparations, the project advances to the embedded testing phase, where the performance and efficiency of the algorithms can be evaluated.

3.6 Chapter Summary

This chapter detailed the entire Design Space Exploration framework, marking the shift from theoretical review to systematic methodology and implementation. The process began by es-

establishing the project requirements and designing the utilised framework, which guided the subsequent selection and configuration of the game environments (Pong, Breakout, and Space Invaders). These environments were chosen to provide increasing levels of complexity, enabling a rigorous and comparative evaluation of the DRL agents.

The implementation involved outlining the neural network architectures and the software framework used for training, followed by the selection and initial testing of the various DRL algorithms to establish performance baselines. The analysis of these algorithms led to the optimisation and adaptation work, aiming to reduce the computational cost and memory footprint of the DRL agents and making them viable for deployment on resource-limited systems.

After detailing the methodology and implementing and optimising the DRL agents for the embedded platform, the system is ready for assessment. The next chapter will present the evaluation and analysis of the obtained results, measuring the performance of the optimised models and providing a detailed analysis of the resulting trade-offs in game scores, execution times, and resource consumption.

Chapter 4

Evaluation on the Embedded Device

The purpose of this chapter is to evaluate the performance of the selected algorithms once they are deployed on the embedded device. Following the development and optimisation stage presented in the previous chapter, this stage now focusses on evaluating the efficiency, resource usage, and effectiveness under resource constrained settings. By comparing the algorithms, it is possible to identify the trade-offs between computational cost and performance, and determine which approaches are best suited for embedded reinforcement learning.

The evaluation is carried out by considering metrics such as execution time, memory consumption, and hardware utilisation, along with the learning performance of the agents. This enables a better assessment of the quality of the policies and also demonstrates their feasibility on constrained systems.

The structure of this chapter is as follows:

- **Section 4.1** introduces the evaluation methodology, detailing the metrics, procedures, and criteria used to assess each algorithm.
- **Section 4.2** presents the main results of the evaluation, showing the performance of the selected algorithms on the embedded platform and discussing their relative advantages and disadvantages.
- **Section 4.3** explores additional environments tested during the project, providing insights into how the algorithms generalise to different tasks.
- **Section 4.4** provides a chapter summary, concluding the evaluation, summarising the performance and introducing the conclusions to be presented in the final chapter.

4.1 Evaluation Methodology

The evaluation methodology was designed to provide a fair comparison between the selected algorithms. Since the primary objective is to identify reinforcement learning methods that are both effective and feasible for embedded deployment, the analysis considers not only the performance of the agents but also the computational costs and resource efficiency.

The chosen algorithms were evaluated on the NVIDIA Jetson Orin Nano under consistent experimental conditions, ensuring fair comparison across all tests. The three previously selected environments, *Pong*, *Breakout*, and *Space Invaders* were used to evaluate each algorithm across increasing levels of complexity and computational demands. Training was conducted for 2500 episodes per algorithm, with minor hyperparameter adjustments applied when necessary.

Learning performance was assessed through the evolution of average scores, maximum scores achieved, and the overall stability of the learning curves. Computational efficiency was measured by recording execution times and monitoring hardware utilisation, including CPU load, GPU load, and RAM consumption.

The evaluation process considers not only isolated metrics but also the trade-offs between them. An algorithm capable of achieving high scores but requiring excessive time or resources may not be suitable for embedded devices, while a lighter algorithm with moderate scores could represent a more promising solution.

4.2 Results on Selected Environments

The evaluation process begins by presenting the results obtained across the selected environments, each representing a different level of difficulty and computational demand. For each environment, the chosen algorithms were trained and evaluated. Given the memory constraints of the target platform, the replay buffer size for value-based algorithms was reduced from 100,000 to 50,000 experiences, nearly halving RAM usage and ensuring that training could proceed without causing the device to shut down. The results were then shown through learning curves, score logs, and comparison tables, followed by an analysis of the performance and resource trade-offs observed.

4.2.1 Low Complexity Environment - Pong

The *Pong* environment provides a lightweight benchmark that makes it possible to evaluate how quickly and efficiently algorithms can learn a basic control task with a reduced action pool under embedded constraints. It serves as a starting point for assessing the embedded performance of the selected methods before moving on to more complex environments.

To accelerate training without affecting the fundamental learning process, the number of points required to win a game was reduced from 21 to 11 points (by adjusting the game's settings). In addition, the learning curves for *Pong* contain two distinct moving averages, a blue one and a red one, to measure the agent's and the opponent's performance, respectively.

The PPO curve (Figure 4.1) showed very limited improvement, with the agent never winning a game during training. The average scores remained close to 0 (Listing 4.1), and the best result was only 4 points, reflecting the struggle of policy-based algorithms with environments that feature sparse reward signals.

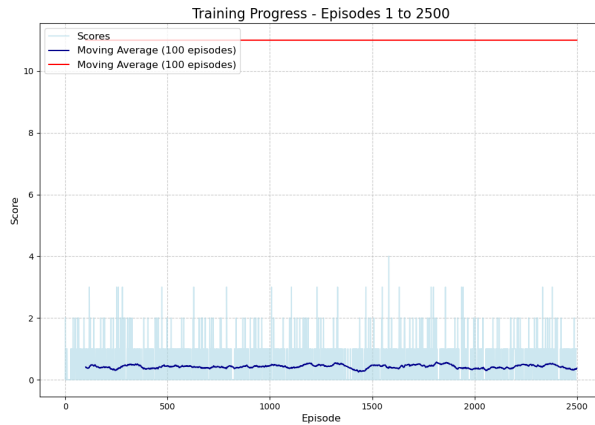


Figure 4.1 Pong Learning Curve with PPO

```

Episode 1 Average Score:   Agent - 0.00 x 11 - Opponent
Episode 250 Average Score: Agent - 0.32 x 11 - Opponent
Episode 500 Average Score: Agent - 0.45 x 11 - Opponent
Episode 750 Average Score: Agent - 0.42 x 11 - Opponent
Episode 1000 Average Score: Agent - 0.45 x 11 - Opponent
Episode 1250 Average Score: Agent - 0.47 x 11 - Opponent
Episode 1500 Average Score: Agent - 0.47 x 11 - Opponent
Episode 1750 Average Score: Agent - 0.46 x 11 - Opponent
Episode 2000 Average Score: Agent - 0.36 x 11 - Opponent
Episode 2250 Average Score: Agent - 0.49 x 11 - Opponent
Episode 2500 Average Score: Agent - 0.37 x 11 - Opponent

```

```

Script Runtime: 4 hours 16 minutes
First Win: Agent Never Won
Best Score: Agent - 4 x 11 - Opponent (Episode 1582)

```

Listing 4.1 Pong Average Scores with PPO

The DQN agent (Figure 4.2) achieved better results, showing modest but steady improvements. The agent managed to score up to 8 points in a single game (Listing 4.2), but failed to achieve a single win against the opponent. The convergence remained shallow, with the average scores around 2 points.

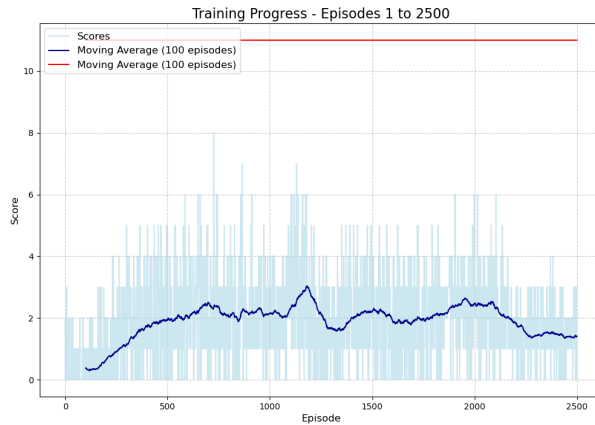


Figure 4.2 Pong Learning Curve with DQN

```

Episode 1 Average Score: Agent - 0.00 x 11 - Opponent
Episode 250 Average Score: Agent - 0.90 x 11 - Opponent
Episode 500 Average Score: Agent - 1.94 x 11 - Opponent
Episode 750 Average Score: Agent - 2.34 x 11 - Opponent
Episode 1000 Average Score: Agent - 2.16 x 11 - Opponent
Episode 1250 Average Score: Agent - 2.11 x 11 - Opponent
Episode 1500 Average Score: Agent - 2.22 x 11 - Opponent
Episode 1750 Average Score: Agent - 1.98 x 11 - Opponent
Episode 2000 Average Score: Agent - 2.47 x 11 - Opponent
Episode 2250 Average Score: Agent - 1.52 x 11 - Opponent
Episode 2500 Average Score: Agent - 1.40 x 11 - Opponent

```

```

Script Runtime: 4 hours 29 minutes
First Win: Agent Never Won
Best Score: Agent - 8 x 11 - Opponent (Episode 727)

```

Listing 4.2 Pong Average Scores with DQN

The DDQN implementation (Figure 4.3) stood out as a very effective method for this environment. The learning curve demonstrated continuous progress, with the agent achieving its first win around halfway through training, and later reaching an almost perfect game of 11 to 1 (Listing 4.3). The average scores were near 10 points at the end of training, demonstrating that the algorithm was able to consistently outperform the opponent. The significantly longer runtimes were mainly due to the competitiveness of the agent at the end of training, with each episode having an increased amount of points for both the agent and the opponent.

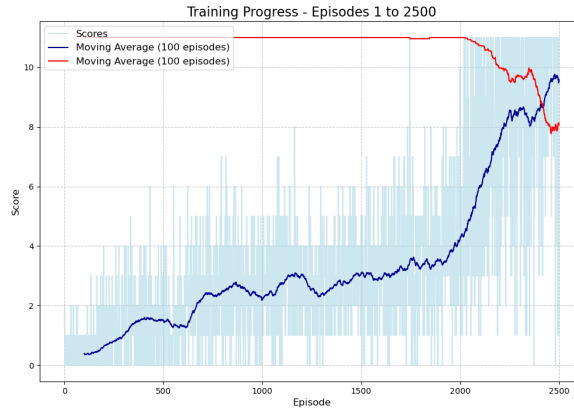


Figure 4.3 Pong Learning Curve with DDQN

```

Episode 1 Average Score:   Agent - 0.00 x 11 - Opponent
Episode 250 Average Score: Agent - 0.80 x 11 - Opponent
Episode 500 Average Score: Agent - 1.51 x 11 - Opponent
Episode 750 Average Score: Agent - 2.34 x 11 - Opponent
Episode 1000 Average Score: Agent - 2.19 x 11 - Opponent
Episode 1250 Average Score: Agent - 2.56 x 11 - Opponent
Episode 1500 Average Score: Agent - 3.10 x 11 - Opponent
Episode 1750 Average Score: Agent - 3.58 x 10.96 - Opponent
Episode 2000 Average Score: Agent - 4.26 x 10.73 - Opponent
Episode 2250 Average Score: Agent - 8.56 x 9.54 - Opponent
Episode 2500 Average Score: Agent - 9.59 x 8.07 - Opponent

```

```

Script Runtime: 7 hours 42 minutes
First Win: Agent - 11 x 7 - Opponent (Episode 1746)
Best Score: Agent - 11 x 1 - Opponent (Episode 2180)

```

Listing 4.3 Pong Average Scores with DDQN

To better interpret these results, a comparison table was compiled (Table 4.1), which summarises the trade-offs between performance and efficiency. The DDQN algorithm achieved the highest average and best scores and was the only one that managed to consistently defeat the opponent, although at the cost of a substantially longer runtime and higher resource usage. The DQN provided a middle ground, offering moderate results in less time, while PPO proved to be the most resource-efficient but did not achieve meaningful performance.

Algorithm	Best Avg. Score	Best Score	Runtime	CPU (%)	GPU (%)	RAM (GB)
PPO	0.49	4	4h16m	25	90	1.5
DQN	2.47	8	4h29m	30	95	3
DDQN	9.59	11	7h42m	30	95	4

Table 4.1 Embedded Device Algorithm Comparison on the Pong Environment

4.2.2 Medium Complexity Environment - Breakout

The *Breakout* environment represents the medium difficulty game, which requires the agent to keep the ball in play, predict its trajectory, and strategically target the bricks that maximise rewards. This added complexity makes it a more informative environment for evaluating the capacity of algorithms to balance reactive control with long-term planning.

The PPO learning curve (Figure 4.4) demonstrated that the algorithm achieved modest performance for a policy-based method, despite showing some learning instability. The algorithm obtained an average score of 8 points and a high score of 23 points (Listing 4.4), which is still below the human average. This algorithm struggled to improve past the middle training stage, meaning its on-policy nature was less effective for the long-term strategic moves required in Breakout.

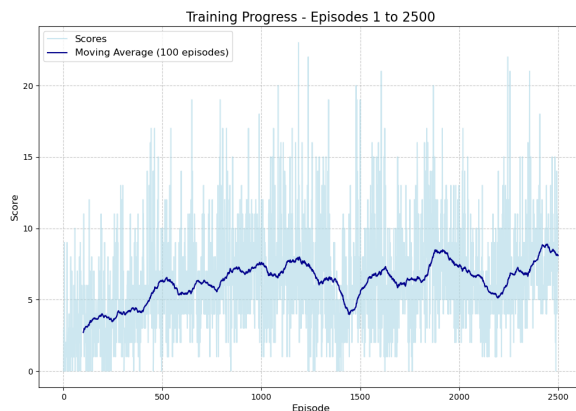


Figure 4.4 Breakout Learning Curve with PPO

```
Episode 1 Average Score: 0.00
Episode 250 Average Score: 3.51
Episode 500 Average Score: 6.37
Episode 750 Average Score: 5.87
Episode 1000 Average Score: 7.56
Episode 1250 Average Score: 7.25
Episode 1500 Average Score: 5.43
Episode 1750 Average Score: 6.35
Episode 2000 Average Score: 7.37
Episode 2250 Average Score: 6.52
Episode 2500 Average Score: 8.08
```

```
Script Runtime: 2 hours 52 minutes
Highest Score: 23.0 (Episode 1189)
```

Listing 4.4 Breakout Average Scores with PPO

The DQN curve (Figure 4.5) showed much more effective learning, with a continuous and steep upward trend. The average score of almost 17 points and the high score of 37 points (Listing 4.5), demonstrate that the agent can achieve above human average results multiple times. The use of an off-policy strategy and a replay buffer allowed this algorithm to make better use of sampled experiences, resulting in higher final performances.

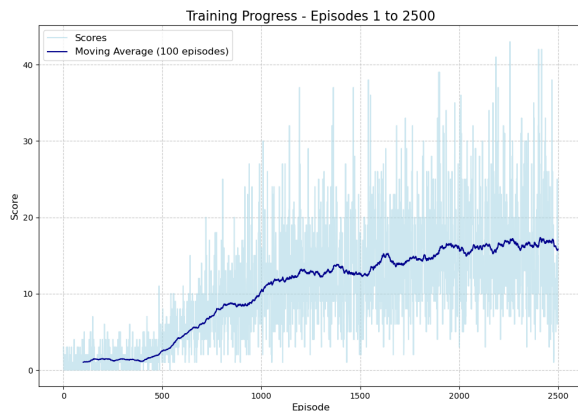


Figure 4.5 Breakout Learning Curve with DQN

Episode 1 Average Score:	0.00
Episode 250 Average Score:	1.21
Episode 500 Average Score:	2.48
Episode 750 Average Score:	7.07
Episode 1000 Average Score:	10.41
Episode 1250 Average Score:	13.02
Episode 1500 Average Score:	12.48
Episode 1750 Average Score:	14.63
Episode 2000 Average Score:	15.99
Episode 2250 Average Score:	16.80
Episode 2500 Average Score:	15.82

Script Runtime: 4 hours 24 minutes
Highest Score: 37.0 (Episode 1365)

Listing 4.5 Breakout Average Scores with DQN

The DDQN learning curve (Figure 4.6) exhibited a very strong performance, showing both rapid convergence and continuous growth. The average scores of over 24 points, show that the agent is consistently getting performances near average human level, which can be further confirmed by its high score of 74 points and the various episodes where it got above 31 points (Listing 4.6). This superior performance is mainly due to the use of the PER buffer, which prioritises the sampling of high-value experiences and reduces the overestimation bias present in the standard DQN.

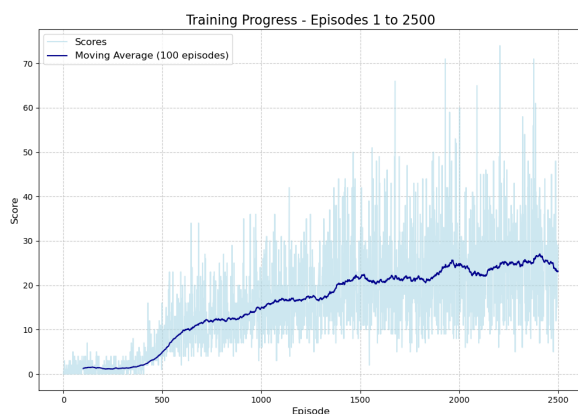


Figure 4.6 Breakout Learning Curve with DDQN

Episode 1 Average Score:	0.00
Episode 250 Average Score:	1.13
Episode 500 Average Score:	4.98
Episode 750 Average Score:	12.03
Episode 1000 Average Score:	14.96
Episode 1250 Average Score:	17.34
Episode 1500 Average Score:	21.57
Episode 1750 Average Score:	21.70
Episode 2000 Average Score:	24.36
Episode 2250 Average Score:	24.70
Episode 2500 Average Score:	23.19

Script Runtime: 4 hours 25 minutes
Highest Score: 74.0 (Episode 2207)

Listing 4.6 Breakout Average Scores with DDQN

The comparison table (Table 4.2) highlights the trade-offs between performance and efficiency. The DDQN clearly outperformed the other algorithms, achieving the highest average and maximum scores, followed by DQN, which also produced decent results with a slower convergence. The PPO method lagged behind in terms of raw performance, but proved substantially more efficient, completing training in less time and with the lowest CPU, GPU, and memory usage. The performance gap between PPO and the off-policy methods in *Breakout*

suggests that for tasks requiring greater strategic depth, the improved sample efficiency and stability of value-based algorithms justifies the higher computational costs.

Algorithm	Best Avg. Score	High Score	Runtime	CPU (%)	GPU (%)	RAM (GB)
PPO	8.08	23.0	2h52m	25	90	1.5
DQN	16.80	37.0	4h24m	30	95	3
DDQN	24.70	74.0	4h25m	30	95	4

Table 4.2 Embedded Device Algorithm Comparison on the Breakout Environment

When comparing these embedded results with the preliminary desktop experiments, clear differences emerge. For the first 1000 episodes, both DQN and DDQN converged slightly slower on the embedded device, while PPO displayed nearly identical behaviour across both platforms. This performance gap can be attributed to the reduced replay buffer size required on the embedded device, a constraint that mostly affected value-based methods, since these rely heavily on experience replay buffers, while leaving the policy-based approach unchanged.

4.2.3 High Complexity Environment - Space Invaders

The *Space Invaders* environment is the most demanding of the three tested games, combining a larger action space with dynamic difficulty escalation as enemies accelerate and attack patterns intensify. This setting requires agents to react quickly and manage longer-term survival strategies under increasingly complex conditions, making it an excellent benchmark to evaluate the adaptability of the selected algorithms.

The PPO curve (Figure 4.7) revealed limited progress, with learning fluctuations and early convergence to suboptimal behaviours. Although the algorithm reached a high score of 775 points, its average performance peaked around 221 points and remained low until the end of training (Listing 4.7), reflecting the challenges policy-based methods face in environments that demand both precision and long-term planning.

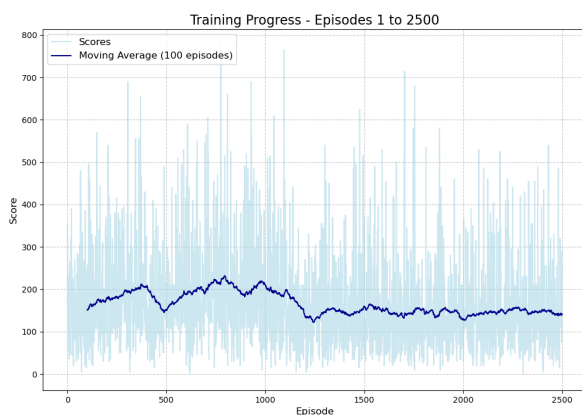


Figure 4.7 Space Invaders Learning Curve with PPO

Episode 1 Average Score:	50.00
Episode 250 Average Score:	182.50
Episode 500 Average Score:	150.15
Episode 750 Average Score:	221.15
Episode 1000 Average Score:	210.20
Episode 1250 Average Score:	126.95
Episode 1500 Average Score:	156.70
Episode 1750 Average Score:	149.80
Episode 2000 Average Score:	129.75
Episode 2250 Average Score:	151.65
Episode 2500 Average Score:	141.15

Script Runtime: 4 hours 21 minutes
Highest Score: 775.0 (Episode 776)

Listing 4.7 Space Invaders Average Scores with PPO

The DQN agent (Figure 4.8) displayed stronger results, showing steady improvements and achieving an average score of 262 points with a maximum of 855 points (Listing 4.8). Its off-policy replay strategy enabled more stable learning, though variability in later episodes indicates that convergence remained somewhat inconsistent.

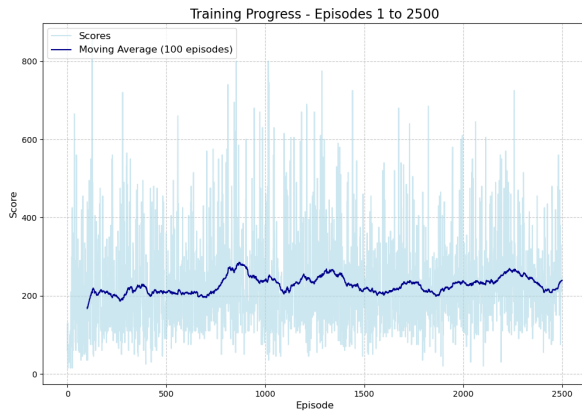


Figure 4.8 Space Invaders Learning Curve with DQN

```

Episode 1 Average Score:      60.00
Episode 250 Average Score:   196.95
Episode 500 Average Score:   208.95
Episode 750 Average Score:   215.95
Episode 1000 Average Score:  240.90
Episode 1250 Average Score:  237.55
Episode 1500 Average Score:  216.20
Episode 1750 Average Score:  231.70
Episode 2000 Average Score:  236.55
Episode 2250 Average Score:  262.35
Episode 2500 Average Score:  239.00

```

```

Script Runtime: 3 hours 58 minutes
Highest Score: 855.0 (Episode 126)

```

Listing 4.8 Space Invaders Average Scores with DQN

The DDQN implementation (Figure 4.9) delivered the best overall performance. It achieved an average score of 322 points and a high score of 930 points (Listing 4.9), reaching close to the average human scores for this environment. The algorithm demonstrated a faster convergence in the early stages of training, with greater stability and more noticeable improvements throughout the episodes.

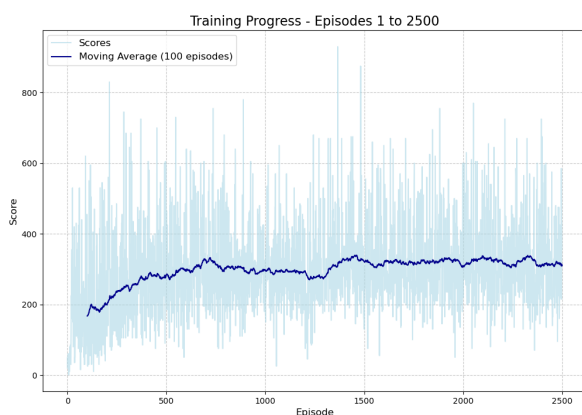


Figure 4.9 Space Invaders Learning Curve with DDQN

```

Episode 1 Average Score:      20.00
Episode 250 Average Score:   234.65
Episode 500 Average Score:   283.60
Episode 750 Average Score:   314.15
Episode 1000 Average Score:  297.10
Episode 1250 Average Score:  275.30
Episode 1500 Average Score:  322.10
Episode 1750 Average Score:  313.50
Episode 2000 Average Score:  310.10
Episode 2250 Average Score:  312.05
Episode 2500 Average Score:  310.00

```

```

Script Runtime: 5 hours 52 minutes
Highest Score: 930.0 (Episode 1367)

```

Listing 4.9 Space Invaders Average Scores with DDQN

To examine these results, the comparison table (Table 4.3) highlights the trade-offs between performance and efficiency. The DDQN once again achieved the best scores by a wide margin, but at the cost of the longest runtime and highest resource usage. The DQN offered a competitive balance, reaching strong performance in a surprisingly short time, while PPO stood

out for its computational efficiency, maintaining the lowest resource usage, but failing to match the performance of the value-based methods. These results indicate that, in highly complex environments such as *Space Invaders*, value-based approaches are better suited, even with increased computational costs.

Algorithm	Best Avg. Score	High Score	Runtime	CPU (%)	GPU (%)	RAM (GB)
PPO	221.15	775.0	4h21m	25	90	1.5
DQN	262.35	855.0	3h58m	30	95	3
DDQN	322.10	930.0	5h52m	30	95	4

Table 4.3 Embedded Device Algorithm Comparison on the Space Invaders Environment

4.2.4 Result Analysis and Discussion

After analysing all three environments, we could recognise some consistent patterns in the behaviour of each algorithm. The PPO implementation was always the lightweight approach, maintaining the lowest CPU, GPU, and RAM usage, making it a promising choice when the priority is minimising computational load rather than maximising performance. However, its learning dynamics often stagnated early, resulting in substantially lower scores compared to the value-based methods.

The DQN proved to be a solid middle ground, achieving reasonable scores with acceptable runtimes and moderate resource consumption. It produced decent results in all environments and in less time than its successor, making it a practical option when the objective is balancing performance and efficiency.

The DDQN consistently delivered the strongest performances, achieving the highest average and maximum scores across all environments. Although it required the longest runtimes and consumed the most resources, its superior stability and effectiveness outweighed these costs.

Overall, PPO remained useful as a lightweight baseline, DQN offered a balanced compromise between performance and computational cost and DDQN stood out as the most suitable choice to maximise learning effectiveness. For this project’s specific embedded device, running DDQN proved entirely feasible, and its superior performance clearly justified the additional computational demands.

4.3 Results on Modified Environments

Beyond the core evaluation, several exploratory experiments were conducted to investigate how reinforcement learning agents behave under alternative conditions. These experiments are not intended as a primary benchmark, but rather as an opportunity to explore interesting extensions of the main work and to identify potential routes for future research.

4.3.1 Alternative Game Modes for Breakout

The *Breakout* environment includes gameplay variations such as *Catch*, where the paddle momentarily holds the ball upon contact, and *Breakthrough*, where the ball passes through bricks instead of bouncing back. These variants significantly alter the game dynamics, requiring different strategies to maximise rewards. The objective of this experiment was to test whether a model previously trained on the standard *Breakout* could provide an advantage when adapting to these alternative modes.

For each game mode, two DDQN agents were trained for 1000 episodes. The first started entirely from scratch, without any prior experience, while the second was initialised with a previously trained model on the standard *Breakout*. This setup made it possible to assess whether prior knowledge could accelerate adaptation or lead to improved performance in the alternative game modes.

The results obtained in the *Catch* variation (Listing 4.10) reveal a clear advantage for the agent initialised with prior experience from the standard *Breakout*. The agent trained from scratch required several hundred episodes before showing meaningful improvements, while the previously trained agent started at a much higher baseline and consistently outperformed the untrained agent throughout the experiment. The trained model achieved an average score of around 18 points compared to 13.5 for the agent without prior experience, and a substantially better high score of 45, compared to the inexperienced agent's 26 points. These findings indicate that prior knowledge can be effectively transferred across similar game modes, accelerating adaptation and leading to higher overall performances.

	Without Experience	With Experience
Episode 1 Average Score :	0.00	6.0
Episode 100 Average Score :	1.13	5.65
Episode 200 Average Score :	1.46	7.23
Episode 300 Average Score :	1.35	8.61
Episode 400 Average Score :	1.51	10.76
Episode 500 Average Score :	3.17	12.58
Episode 600 Average Score :	7.43	13.38
Episode 700 Average Score :	10.66	15.64
Episode 800 Average Score :	11.91	17.87
Episode 900 Average Score :	13.72	18.13
Episode 1000 Average Score :	13.53	17.99
Highest Score :	26.0 (Episode 684)	45 (Episode 703)

Listing 4.10 Breakout *Catch* Average Scores

In the *Breakout Breakthrough* game mode (Listing 4.11), both agents achieved considerably higher scores compared to the standard environment due to the new brick clearing mechanic of the mode. The agent trained without prior experience showed steady improvement, achieving almost 200 points by the end of training, with a highest score of 325 points. The agent initialised with knowledge from the standard *Breakout* environment began with a substantial advantage, starting at an average score of 120 points and maintaining a lead throughout training,

reaching an average score of 209 points and a maximum score of 338 points. These results indicate that prior training on the base *Breakout* environment provided a measurable head start in this mode, accelerating convergence and enabling consistently stronger performance.

	Without Experience	With Experience
Episode 1 Average Score :	0.00	120.00
Episode 100 Average Score :	70.76	110.42
Episode 200 Average Score :	58.61	129.22
Episode 300 Average Score :	62.20	125.48
Episode 400 Average Score :	71.28	143.39
Episode 500 Average Score :	86.71	159.72
Episode 600 Average Score :	119.64	162.72
Episode 700 Average Score :	158.63	172.58
Episode 800 Average Score :	196.44	193.97
Episode 900 Average Score :	198.03	205.38
Episode 1000 Average Score :	195.93	209.30
Highest Score :	325.0 (Episode 873)	338.0 (Episode 891)

Listing 4.11 Breakout *Breakthrough* Average Scores

4.3.2 Cross-Environment Knowledge Transfer

Since the previous experiment tested trained models in alternative game modes, it was also relevant to investigate whether we could transfer knowledge across entirely different game environments. For this, a model trained on *Breakout* was deployed directly on a *Pong* and *Space Invaders* environment. Unlike the game mode variations, however, these environments differ not only in reward structures and gameplay dynamics but also in their parameter definitions, including action pools and state representations. As a result, direct transfer was not feasible, as the trained model was incompatible with the other environments. These observations emphasise that, despite belonging to the same Atari family, each environment requires its own tailored training process unless more advanced adaptation techniques are applied.

4.4 Chapter Summary

This chapter concluded the testing phase by evaluating the selected DRL algorithms on the target platform. The assessment began by establishing the evaluation methodology, which defined the main performance metrics. The results were then presented, comparing the performance of the algorithms across the three game environments, demonstrating their relative strengths and weaknesses under the device's constraints.

The analysis provided insights into the algorithms' suitability for embedded devices and highlighted the trade-offs introduced by the implementation of optimisation techniques. The evaluation also confirms whether the resource limitations of the embedded device were successfully mitigated while still maintaining acceptable game performance.

The evaluation presented in this chapter serves to answer the main questions of this work by validating and quantifying the feasibility of deploying optimised DRL agents on a resource-constrained device. The next chapter will summarise the main contributions of this entire thesis, drawing final conclusions and proposing directions for extending this work.

Chapter 5

Conclusions and Future Work

This chapter summarises the main contributions of the thesis, reflecting on the objectives that guided the investigation, the results achieved, and the challenges encountered along the way. It also outlines possible directions for future work, highlighting improvements, extensions, and opportunities for further research.

5.1 Research Recap

The primary goal of this research was to evaluate the deployment of DRL algorithms on embedded devices. To achieve this, a set of algorithms were selected, covering policy-based, value-based, and actor–critic families. These methods were implemented, profiled, and compared in a desktop environment to establish preliminary results and identify promising candidates for embedded deployment.

A variety of optimisation techniques were introduced to adapt the selected algorithms for embedded execution, including changes in data representation, hyperparameter tuning, and the use of mixed precision operations. The algorithms were deployed and evaluated on a Jetson Orin Nano across three Atari environments of increasing difficulty. The evaluation then considered both learning performance and resource consumption, allowing for a comparison of trade-offs between computational efficiency and training effectiveness.

5.2 Achievements

The results demonstrated that the execution of DRL algorithms on embedded platforms is both feasible within a reasonable time (taking a few hours to run 2500 episodes) and efficient, consistently achieving above human average scores, when the right selection of algorithms and optimisation techniques are applied.

Among the tested methods, DDQN delivered the strongest overall performance, PPO proved to be the most lightweight and resource-efficient, and DQN offered a reliable middle ground between the two. These results show that the most suitable algorithm for embedded deployment depends on the desired balance between accuracy and efficiency.

In addition, the project successfully established a reusable framework for training, testing,

and deploying DRL algorithms. This included a structured implementation with agents, models, environments, wrappers, utilities, and support for saving and reusing trained models.

5.3 Limitations

Several limitations were identified during the course of this work, reflecting both implementation constraints and methodological challenges. Since the main objective was to evaluate DRL algorithms on a resource-constrained device, compromises had to be made at multiple stages of development to achieve this goal.

The first challenge was the limited documentation and support available for some of the chosen environments, which complicated their integration and experimentation. This lack of detailed guidance made it more difficult to modify certain environments.

Another limitation encountered was the absence of reference implementations for some algorithms, which requires utilising approximate or adapted versions. These implementations allowed experimentation, but sometimes raised doubts about whether the results reflected the intended design and performance.

Finally, due to the hardware constraints of the embedded device, in terms of both memory and computational capacity, the replay buffer sizes and batch configurations had to inevitably be reduced, which limits the performance of the methods compared to desktop implementations. The use of parallel executions had to be dismissed due to the limited available memory. Furthermore, the long execution times of embedded training made experimentation more challenging, particularly when attempting longer training runs or detailed hyperparameter tuning that could further enhance performance.

5.4 Future Work

Future research could extend this work in several directions. From an algorithmic perspective, exploring more advanced or recently proposed DRL methods could provide better insights into whether newer techniques can outperform the ones evaluated in this project.

Further optimisation strategies should also be explored, such as precise hyperparameter adjustments, systematic tuning, and pruning of redundant parameters could help achieve additional performance. Additionally, implementing lightweight CNNs specifically designed for embedded devices, combined with more aggressive techniques such as quantisation and sparse representations, could significantly reduce computational cost while maintaining accuracy.

Finally, this work provides a foundation for those who wish to continue research in this area. Future studies could focus on deeper comparisons of the most promising algorithms, integrate more advanced optimisation techniques, or implement parallel execution setups to make embedded agents learn faster. These directions contribute to developing more efficient solutions for Deep Reinforcement Learning on embedded devices.

Bibliography

- [1] Agarwal, P., Rahman, A. A., St-Charles, P.-L., Prince, S. J. D., and Kahou, S. E. (2023). Transformers in reinforcement learning: A survey. *ArXiv*, abs/2307.05979.
- [2] Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J. I., Fadhel, M. A., Al-Amidie, M., and Farhan, L. (2021). Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8.
- [3] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.
- [4] Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 449–458. JMLR.org.
- [5] Bhatnagar, S. (2023). *The Reinforce Policy Gradient Algorithm Revisited*. Institute of Electrical and Electronics Engineers.
- [6] Bishop, C. (2006). *Pattern recognition and machine learning*, volume 4. Springer New York.
- [7] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Lechevallier, Y. and Saporta, G., editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg. Physica-Verlag HD.
- [8] Burhanuddin, M. (2023). Efficient hardware acceleration techniques for deep learning on edge devices: A comprehensive performance analysis. *KHWARIZMIA*, 2023:1–10.
- [9] Chapelle, O., Scholkopf, B., and Zien, Eds., A. (2009). Semi-supervised learning (chapelle, o. et al., eds.; 2006) [book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542.
- [10] Dash, A., Ye, J., and Wang, G. (2024). A review of generative adversarial networks (gans) and its applications in a wide variety of disciplines: From medical to remote sensing. *IEEE Access*, 12:18330–18357.
- [11] De Asis, K., Hernandez-Garcia, J. F., Holland, G. Z., and Sutton, R. S. (2018). Multi-step reinforcement learning: a unifying algorithm. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press.

- [12] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, A., Senior, A., Tucker, P., Yang, K., and Ng, A. (2012). Large scale distributed deep networks. *Advances in neural information processing systems*.
- [13] Dong, G., Tang, M., Yan, R., Mu, Z., Cai, L., and Park, B. (2023). *Deep Learning for Autonomous Vehicles and Systems*, pages 9–47. River Publishers.
- [14] Ekundayo, O. and Ezugwu, A. (2024). *Deep Learning: Historical Overview from Inception to Actualization, Models, Applications and Future Trends*. TechRxiv.
- [15] Fan, J. (2023). A review for deep reinforcement learning in atari: benchmarks, challenges, and solutions.
- [16] Feinberg, V., Wan, A., Stoica, I., Jordan, M. I., Gonzalez, J. E., and Levine, S. (2018). Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning. *arXiv e-prints*, page arXiv:1803.00101.
- [17] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2018). Noisy networks for exploration. In *International Conference on Learning Representations*.
- [18] Foundation, F. (2025a). Breakout - arcade learning environment (ale). Online.
- [19] Foundation, F. (2025b). Pong - arcade learning environment (ale). Online.
- [20] Foundation, F. (2025c). Space invaders - arcade learning environment (ale). Online.
- [21] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- [22] Ha, D. and Schmidhuber, J. (2018). World models. *CoRR*.
- [23] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR.
- [24] Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*.
- [25] Hasselt, H. v., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI'16*, page 2094–2100. AAAI Press.
- [26] He, X., Xue, F., Ren, X., and You, Y. (2021). Large-scale deep learning optimizations: A comprehensive survey.

- [27] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press.
- [28] Hribar, J. and Dusparic, I. (2022). Enabling deep reinforcement learning on energy constrained devices at the edge of the network. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 2547–2552.
- [29] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.
- [30] Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).
- [31] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.
- [32] Le, M. T., Wolinski, P., and Arbel, J. (2023). Efficient neural networks for tiny machine learning: A comprehensive review. *CoRR*, abs/2311.11883.
- [33] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–44.
- [34] Li, Z., Samanta, A., Li, Y., Soltoggio, A., Kim, H., and Liu, C. (2023). R^3 : On-device real-time deep reinforcement learning for autonomous robotics.
- [35] Lim, J. and Sung, K. (2007). *FNN (Feedforward Neural Network) Training Method Based on Robust Recursive Least Square Method*. Springer Berlin Heidelberg, Berlin, Heidelberg.
- [36] Malhotra, K. and Kumar, Y. (2020). Challenges to implement machine learning in embedded systems. In *2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 477–481.
- [37] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org.
- [38] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602.

- [39] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.
- [40] Naeem, M., Rizvi, S., and Coronato, A. (2020). A gentle introduction to reinforcement learning and its application in different fields. *IEEE Access*, 8:209320–209344.
- [41] Nair, V. and Hinton, G. E. (2010). *Rectified linear units improve restricted boltzmann machines*. ICML'10. Omnipress, Madison, WI, USA.
- [42] Ng, A. (2004). Feature selection, l_1 vs. l_2 regularization, and rotational invariance. *Proceedings of the Twenty-First International Conference on Machine Learning*.
- [43] Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press.
- [44] NVIDIA Corporation (2023). Nvidia jetson orin nano developer kit. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/>. Accessed: 2025-08-21.
- [45] Rani, V., Nabi, S. T., Kumar, M., et al. (2023). Self-supervised learning: A succinct review. *Archives of Computational Methods in Engineering*, 30:2761–2775.
- [46] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.
- [47] Rummery, G. and Niranjan, M. (1994). On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*.
- [48] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv: Learning*.
- [49] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.
- [50] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.
- [51] Schulman, J., Levine, S., Moritz, P., Jordan, M., and Abbeel, P. (2015). Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15*, page 1889–1897. JMLR.org.
- [52] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347.

- [53] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815.
- [54] Sliwa, B., Piatkowski, N., and Wietfeld, C. (2020). Limits: Lightweight machine learning for iot systems with resource limitations.
- [55] Souchleris, K., Sidiropoulos, G. K., and Papakostas, G. A. (2023). Reinforcement learning in game industry—review, prospects and challenges. *Applied Sciences*, 13(4).
- [56] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.
- [57] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.
- [58] Svoboda, F., Nunes, D., Alizadeh, M., Daries, R., Luo, R., Mathur, A., Bhattacharya, S., Silva, J. S., and Lane, N. D. (2021). Resource efficient deep reinforcement learning for acutely constrained tiny{ml} devices. In *Research Symposium on Tiny Machine Learning*.
- [59] Swiechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J. (2022). Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562.
- [60] Sá, G. and Madeira, C. (2024). Deep reinforcement learning in real-time strategy games: a systematic literature review. *Applied Intelligence*, 55.
- [61] Tarbouriech, J., Zhou, R., Du, S. S., Pirota, M., Valko, M., and Lazaric, A. (2021). Stochastic shortest path: Minimax, parameter-free and towards horizon-free regret. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W., editors, *Advances in Neural Information Processing Systems*.
- [62] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ICML'16*, page 1995–2003. JMLR.org.
- [63] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.
- [64] Wu, Z., Sun, S., Wang, Y., Liu, M., Jiang, X., Li, R., and Gao, B. (2024). Knowledge distillation in federated edge learning: A survey. *Discover Computing*.
- [65] Yarats, D., Fergus, R., Lazaric, A., and Pinto, L. (2021). Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv preprint arXiv:2107.09645*.

- [66] Zaheer, R. and Shaziya, H. (2019). A study of the optimization algorithms in deep learning. In *2019 Third International Conference on Inventive Systems and Control (ICISC)*, pages 536–539. Institute of Electrical and Electronics Engineers.
- [67] Zhu, S., Voigt, T., Rahimian, F., and Ko, J. (2024). On-device training: A first overview on existing systems. *ACM Transactions on Sensor Networks*, 20(6):1–39.
- [68] Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning.