

PAVOL JOZEF ŠAFÁRIK UNIVERSITY IN KOŠICE
FACULTY OF SCIENCE

**OBJECT DETECTION AND
BEHAVIOR OPTIMIZATION OF
AGENTS IN COMPUTER GAME**

DIPLOMA THESIS

Field of Study:	Informatics
Institute:	Institute of Computer Science
Tutor:	Ľubomír Antoni
Counsellor:	Ing. Matúš Semančík

Košice 2026
Roland Marek Rak

Acknowledgments

I would like to thank my diploma thesis supervisor Lubomír Antoni for his valuable comments during the development of my diploma thesis.



P. J. Šafárik University in Košice
Faculty of Science

THESIS ASSIGNMENT

Name and Surname: Bc. Roland Marek Rak
Study programme: Informatics (Single degree study, master II. deg., Full-Time form)
Field of Study: Computer Science
Type of Thesis: Diploma thesis
Language of Thesis: English
Secondary language: Slovak

Title: Object detection and behavior optimization of agents in computer game
Title SK: Detekcia objektov a optimalizácia správania agentov v počítačovej hre
Aims:
1. Spracovať prehľad metód učenia posilňovaním (reinforcement learning, RL) používané v prostredí počítačových hier (napr. Q-learning, Proximal Policy Optimization) a metód na detekciu objektov v prostredí hier.
2. Modifikovať existujúce metódy detekcie objektov a učenia posilňovaním pre efektívnejšiu optimalizáciu správania agentov vo vybranej počítačovej hre.
3. Porovnať výkon agenta na týchto metódach z hľadiska metrík, ako sú skóre, efektivita učenia a iné.
References:
1. Shakyia, A. K., Pillai, G., & Chakrabarty, S. (2023). Reinforcement learning algorithms: A brief survey. *Expert Systems with Applications*, 231, 120495.
2. Krichen, M. (2023, July). Deep reinforcement learning. In *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (pp. 1-7). IEEE.

Supervisor: doc. RNDr. Ľubomír Antoni, PhD.
Consultant: Ing. Matúš Semančík
Opponent: Ing. Manohar Gowdru Shridhara
Institutes: ÚINF - Institute of Computer Science
Head of Institute: doc. RNDr. Ondrej Krídlo, PhD.

Approved: 30.04.2026

Abstrakt

Táto diplomová práca predstavuje vzorkovo efektívny, hybridný systém vnímania a riadenia pre autonómnu navigáciu a zber surovín v komplexnom 3D virtuálnom prostredí MMORPG hry *World of Warcraft*. Tradičné end-to-end modely hlbokého učenia posilňovaním narážajú na extrémnu vzorkovú neefektívnosť pri súčasnom spracovávaní vysokorozmerných obrazových pixelov a učení sa stratégií správania. Tento problém riešime oddelením vizuálneho vnímania prostredia od optimalizácie stratégie pomocou viacfázovej pipeline.

Najprv nástroj na snímanie okna hry s nízkou latenciou zachytáva grafický buffer v reálnom čase pre doladený jednofázový model na detekciu objektov YOLOv8. Model je natrénovaný na vlastnom resete dát rozšírenom o geometrické posuny, Gaussov šum a zmeny jasu pre zabezpečenie robustnosti voči zmenám prostredia. Táto vrstva počítačového videnia komprimuje surové grafické vstupy do hustého, 19-rozmerného stavového vektora, ktorý obsahuje relatívne súradnice, veľkosti a parametre sledovania cieľov a hrozieb. Tieto štruktúrované vektory následne spracováva stratégia Deep Q-Network (DQN) v rámci na mieru navrhnutého prostredia Gymnasium. Agent optimalizuje svoje pohybové trajektórie na základe kontextovo orientovanej trojfázovej funkcie odmien (potulovanie, lov, zber).

Empirické výsledky ukazujú, že model dosahuje stabilizáciu správania za menej ako 1 000 krokov. Natrénovaný agent úspešne eliminuje zbytočné otáčanie, vyhýba sa hrozbám a využíva nízkoúrovňovú emuláciu DirectInput na presné klikanie a zber surovín bez invazívnych zásahov do pamäte hry. Tento prístup potvrdzuje funkčnosť neinvazívnej alternatívy pre automatizáciu v komplex-

ných virtuálnych svetoch v reálnom čase.

Kľúčové slová: *Hlboké učenie posilňovaním, Detekcia objektov, YOLOv8, Deep Q-Network, Počítačové hry, Vlastné prostredie Gymnasium, Autonómna navigácia.*

Abstract

This thesis presents a sample-efficient, hybrid perception-action framework for autonomous navigation and resource gathering in complex 3D virtual environments using the MMORPG *World of Warcraft*. Traditional end-to-end deep reinforcement learning models suffer from severe sample inefficiency when simultaneously processing high-dimensional screen pixels and learning behavioral strategies. We address this bottleneck by decoupling visual perception from policy optimization using a multi-stage pipeline.

First, a low-latency window interception utility captures real-time frame buffers for a fine-tuned, single-stage YOLOv8 object detection model. This model is trained on a custom dataset augmented with geometric translations, Gaussian noise, and brightness shifts to ensure environmental robustness. The computer vision layer compresses raw graphical streams into a dense, 19-dimensional object-centric state vector containing relative coordinates, sizes, and tracking parameters of targets and threats. Downstream, a Deep Q-Network (DQN) policy inside a custom Gymnasium environment utilizes these structured vectors for decision-making. Guided by a context-driven, three-phase reward function (Wandering, Hunting, Collecting), the agent learns to optimize its locomotion trajectories.

Empirical results demonstrate steady behavioral stabilization in fewer than 1,000 steps. The trained agent successfully eliminates redundant rotational movements, bypasses active hazards, and executes low-level OS input injection via DirectInput emulations to harvest resource nodes without invasive memory manipulation. This approach provides a sample-sparse, non-invasive alternative for real-time automation in complex virtual worlds.

Keywords: *Deep Reinforcement Learning, Object Detection, YOLOv8, Deep Q-Network, Computer Games, Custom Gymnasium Environment, Autonomous Navigation.*

Contents

Introduction	12
1 Theoretical Background	14
1.1 Artificial Intelligence and Machine Learning	14
1.2 Neural Networks and Deep Learning	15
1.3 Reinforcement Learning	17
1.4 Computer Vision	18
1.5 Object Detection	19
1.6 Reinforcement Learning in Games	21
2 Related Work	23
2.1 Vision-Based Reinforcement Learning in Virtual Environments	23
2.2 Object Detection Assisted Reinforcement Learning	24
2.3 Autonomous Navigation and Behavior Optimization in Games	25
2.3.1 Positioning of This Work	26
3 Deep Learning Foundations and Neural Architectures	27
3.1 Biological Inspiration	27
3.2 Artificial Neuron	29
3.3 Perceptron	29
3.4 Multi-layer Perceptron	31
3.5 Forward Propagation	32
3.6 Backpropagation	33
3.7 Activation Functions	34
3.8 Loss Functions	35
3.9 Optimization (Gradient Descent, Adam)	36
3.10 Convolutional Neural Networks	37

4	Reinforcement Learning Frameworks and Policy Optimization	39
4.1	Markov Decision Process	39
4.1.1	State	40
4.1.2	Action	41
4.1.3	Reward	41
4.1.4	Policy	41
4.1.5	Value Function	42
4.2	Bellman Equation	42
4.3	Q-learning	43
4.4	Deep Q Network	44
4.5	Double DQN	45
4.6	Proximal Policy Optimization	46
4.7	Comparison of Paradigms	47
5	Object Detection and Computer Vision Pipelines	50
5.1	Computer Vision Pipeline	50
5.2	Feature Extraction	51
5.3	Image Classification	52
5.4	Localization	52
5.5	Detection	53
5.6	R-CNN Family	54
5.7	YOLO	56
5.8	Why Object Detection is Important for RL Agents	58
6	Environment Design	60
6.1	Simulation Environment (Pygame Minimap Proof-of-Concept)	60
6.2	Live Game Environment (World of Warcraft Integration)	62
6.3	State Representation and Entity Properties	63
7	System Architecture	64
7.1	The Hybrid Perception-Action Loop	64
7.2	YOLO Interface and State Formatter	65
7.3	Policy Network Ingestion Pipeline	66

8	Implementation	67
8.1	Environment Interaction API (Custom Gymnasium Wrapper)	67
8.2	Reward Function Engineering	68
8.3	Locomotion and OS-Level Input Injection	70
8.4	Hyperparameter Configurations	71
9	Experiments and Results	72
9.1	Dataset Generation and Augmentation Analysis	72
9.2	Computer Vision Model Performance Evaluation	76
9.3	Reinforcement Learning Trajectory and Convergence	79
9.3.1	Phase 1: Proof-of-Concept Simulation Training	79
9.3.2	Phase 2: Live In-Game Deployment Results	79
9.4	Evaluation of Thesis Objectives and Agent Performance	81
10	Discussion	85
10.1	Interpretation of Core Findings	85
10.2	Systemic Constraints and Real-Time Latency Challenges	86
10.3	Comparative Analysis and Architectural Trade-offs	87
10.4	Future Architectural Improvements	90
11	Conclusion	91

List of Figures

3.1	The components of a typical neurone	28
5.2	Yolo pipeline (simplified)	58
6.3	2D simulation (Proof of concept).	62
9.4	Segmented training subset samples showcasing localized bounding box annotations configured for the HUD minimap tracking class.	73
9.5	Annotated image frames showing ground-truth bounding box assignments across distinct 3D environmental entities used to assemble the custom dataset.	73
9.6	Original asset frame	74
9.7	Geometric translation matrix	75
9.8	Gaussian blur degradation	75
9.9	Gaussian noise injection	76
9.10	YOLOv8 training mosaic (Batch 0) illustrating annotated bounding boxes, object scales, and mixed background textures used during the optimization phase.	77
9.11	Ground-truth validation labels (Batch 0)	78
9.12	Model inference predictions (Batch 0)	78
9.13	Side-by-side comparison of validation batch 0, demonstrating tight bounding-box matching and highly confident localization against ground-truth labels.	78
9.14	Live environment training progression profiles tracking mean episodic rewards, loss profiles, and survival step lengths over the training duration.	80
9.15	The real-time operational perception loop of the active agent, capturing both localized map vectors and unconstrained 3D view coordinates.	82
9.16	Active collection phase execution, demonstrating an exact absolute cursor placement to trigger the environmental interaction loop.	83

List of Tables

4.1	Comparative Analysis of Reinforcement Learning Paradigms	48
8.1	Hyperparameter layout for the integrated YOLOv8 feature extractor and stable-baselines3 DQN agent.	71
10.1	Comparative Matrix of the Proposed Pipeline Components versus Al- ternative Implementations	89

Introduction

Modern video games provide massive, dynamic virtual environments. These worlds serve as excellent testing grounds for artificial intelligence. Navigating a 3D environment requires an agent to make complex, real-time decisions. Traditional game automation relies heavily on hard-coded rules or direct memory reading. However, hard-coded rules are fragile and fail when conditions change. On the other hand, reading game memory can violate terms of service and lacks human-like visual awareness.

To overcome these issues, researchers have turned to deep reinforcement learning (RL). This approach allows autonomous agents to learn behaviors directly through trial and error. Unfortunately, standard end-to-end reinforcement learning has a major flaw. It is incredibly sample-inefficient. Forcing a network to learn how to see and how to move at the same time requires millions of operational training steps. This constraint makes live training inside an active, real-time video game client nearly impossible due to execution delays and rendering overhead.

This thesis solves the sample-efficiency problem by decoupling visual perception from behavioral control. We implement a hybrid framework that splits the task into two distinct components. First, a computer vision layer uses a fine-tuned YOLOv8 model to process the raw screen buffer. This model detects environmental objects, target resources, and potential threats in real time. Second, the system converts these high-dimensional bounding box coordinates into a dense, object-centric state vector. This clean, low-dimensional vector is then fed directly into a downstream Deep Q-Network (DQN).

By filtering out background graphical noise, the reinforcement learning agent no longer needs to learn how to interpret complex pixel matrices. Instead, it focuses entirely on optimizing its movement, target selection, and collision avoidance behaviors.

We evaluate this hybrid framework in two distinct deployment phases:

1. **A Controlled 2D Simulation:** A custom-designed sandbox built in Pygame.

This environment provides a fast, low-overhead platform to verify our action space and multi-phase reward mechanics.

2. **A Live 3D Game Client:** Direct integration with *World of Warcraft: Mists of Pandaria*. The system captures the window frame buffer, runs real-time inference, and emulates hardware-level keystrokes to navigate and harvest resources autonomously.

The final empirical results demonstrate that our decoupled approach achieves high training efficiency. The agent stabilizes its navigation paths and reaches positive episodic rewards in fewer than 1,000 live steps. Ultimately, this research provides a robust, non-invasive method for video game automation. It demonstrates the immense value of combining single-stage object detectors with object-centric reinforcement learning.

Chapter 1

Theoretical Background

This chapter provides a high-level overview of the core fields that underpin this thesis. Each concept introduced here will be explored in greater technical depth in the chapters that follow. The aim is to establish a shared vocabulary and situate the work within its broader scientific context, covering artificial intelligence, machine learning, neural networks, computer vision, object detection, and the application of reinforcement learning in game environments [1].

1.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) is a broad field of computer science. It focuses on building systems capable of performing tasks that typically require human intelligence, such as reasoning, planning, perception, and decision-making [2]. Early AI systems relied on hand-crafted rules and decision trees designed by domain experts. Although effective in well-defined closed domains such as chess, these approaches proved to be brittle in the noisy, high-dimensional environments found in modern computer vision and open-world games [2].

To circumvent the core limitations of hand-coded logic, the computer science community shifted toward the paradigm of Machine Learning (ML). Machine learning represents a specialized subfield of artificial intelligence centered on the mathematical formulation of statistical algorithms. These algorithms systematically and automatically improve their operational performance over time through continuous exposure to empirical data, completely eliminating the need for explicit step-by-step programmatic intervention [2]. Instead of manually programming a static solution loop, an ML practitioner designs an optimization framework. This framework dynamically adjusts

internal algorithmic parameters by mapping out statistical structures, correlations, and regularities hidden within the incoming training data streams. This data-driven methodology allows the software to discover complex mapping patterns. These patterns are far too intricate or dynamic for a human programmer to identify or formally define using traditional hard-coded conditions.[2].

In contemporary machine learning taxonomy, algorithmic strategies are broadly classified into three fundamental learning paradigms, segregated by the structural nature of their feedback signals and data exposure protocols:

- **Supervised Learning:** The learning system is supplied with an explicitly labeled dataset containing paired inputs and their corresponding ground-truth outputs [2]. The primary optimization objective is to approximate a general mathematical mapping function capable of matching novel, previously unseen input patterns to accurate categorical labels (classification) or continuous scalar values (regression).
- **Unsupervised Learning:** The model operates on unlabeled data streams. It is tasked with independently discovering latent structural properties, probability distributions, or dense feature clusters within the data matrix without any human-labeled intervention or external error correction [2].
- **Reinforcement Learning (RL):** Differing fundamentally from both static paradigms, reinforcement learning abandons pre-collected datasets in favor of a dynamic trial-and-error ecosystem [1]. An autonomous agent interacts sequentially with an active environment, selecting exploratory actions and receiving evaluative feedback in the form of scalar rewards or penalties. This framework forces the system to balance long-term goal planning with immediate situational feedback, making it exceptionally suited for controlling navigation, resource harvesting, and survival loops in complex computer games.

1.2 Neural Networks and Deep Learning

Artificial neural networks (ANNs) represent a sophisticated class of computational topologies loosely modeled after the interconnected biological networks of neurons found within organic brains [2]. At their architectural core, these systems are composed of layered arrangements of basic processing nodes, or artificial neurons, engineered to sequentially process data transformations through a directed graph pipeline

[2]. The standard multilayer feedforward setup is organized into an input layer that accepts raw feature parameters, one or more internal hidden layers tasked with projecting data into latent representation spaces, and a terminal output layer that produces the model’s final predictive hypothesis or distribution matrix [2]. Every structural connection between these computational units is parameterized by an adjustable scalar weight and an additive bias term [2]. The neuron processes incoming signals by calculating their dot-product weighted sum, subsequently routing this value through a mathematical, non-linear activation function [2]. The integration of non-linearity is a mathematical necessity, as it enables the system to approximate highly intricate, non-linear continuous target mappings, bypassing the strict geometric limitations of simple linear regression models [2].

The optimization of these network parameters is driven by the backpropagation algorithm. This algorithm uses a chain rule to calculate the partial derivatives of an explicit loss function with respect to all weights and biases in the architecture [2]. By evaluating these partial derivatives, gradient descent optimization algorithms iteratively adjust the values of the internal parameters in the negative direction of the gradient, minimizing model error in successive training steps [2].

Although classical neural network implementations were structurally limited by shallow computational depths, the modern paradigm of Deep Learning is defined by stacking multiple deep hidden layers to automate feature engineering tasks [2]. This structural expansion enables the automatic, hierarchical extraction of increasingly complex data representations directly from unstructured, high-dimensional inputs:

- **Low-Level Layers:** Isolate primitive structural components, such as localized spatial edges, pixel gradients, and elementary color contrasts [2].
- **Mid-Level Layers:** Combine these primitive vectors to map out distinct geometric boundaries, spatial textures, and structural shapes [2].
- **High-Level Layers:** Synthesize geometric textures into semantic classifications, allowing the system to identify entire functional entities or global environmental scenes [2].

This capability for hierarchical representation learning has established deep neural networks as the dominant methodology in complex fields of cognitive automation, including advanced computer vision and natural language processing [2]. In the context

of this thesis, deep networks serve as highly scalable, non-linear function approximators embedded within reinforcement learning loops [3]. This combination allows autonomous agents to safely digest complex raw sensory representations—such as image frame buffers—and map them directly into discrete behavioral choices, bypassing memory storage bottlenecks that make classical tabular optimization methods completely intractable in large or continuous gaming environments [1, 3].

1.3 Reinforcement Learning

Reinforcement learning (RL) represents a distinct computational paradigm within machine learning centered on goal-directed learning through direct environmental interaction [1]. At its operational core, an autonomous entity, known as the agent, is embedded within an external system, termed the environment, with which it interacts over a sequence of discrete time steps [1]. At each discrete step t , the agent observes a numerical representation of the current configuration of the environment, known as the state s_t , and selects a behavioral choice, called an action a_t , from an available action space [1]. In response to this action, the environment transitions to a new subsequent state s_{t+1} and returns a scalar feedback signal, designated as the reward r_{t+1} [1]. The fundamental objective driving agent behavior is to discover a policy π —a formalized mapping or strategy that correlates perceived states with action selection probabilities—that maximizes the expected cumulative return, or discounted sum of rewards, over an extended operational horizon [1].

To model this sequential decision-making process under conditions of environmental uncertainty, reinforcement learning is heavily based on the mathematical framework of Markov Decision Processes (MDPs) [1]. An MDP assumes that the environment satisfies the Markov property, which means that the probability distribution of the next environmental state depends exclusively on the current state and action, remaining entirely independent of the agent’s historical trajectory or past states [1]. Within this framework, algorithms utilize value functions and action-value functions (known as Q-functions) to estimate the long-term cumulative utility of being in a specific state or executing a particular action under a given policy [1]. Classical tabular reinforcement learning methods, such as standard Q-learning, maintain an explicit memory matrix containing these estimated utility values for every individual state-action pair [5]. Although highly effective in small environments with small, discrete sets of possibilities, this tabular approach becomes mathematically unfeasible when

scaled to high-dimensional or continuous state spaces [1].

The modern integration of deep neural networks as powerful non-linear function approximators solved this scaling limitation, establishing the field of deep reinforcement learning (DRL) [3]. Rather than storing values in rigid and exhaustive tables, a deep network is trained to generalize across similar state spaces, mapping raw sensory inputs directly to value estimates or action probabilities [3]. This technological step allowed RL agents to achieve human-level control across high-dimensional benchmarks, such as the Atari 2600 suite, learning directly from raw game screen pixels and scalar score feeds [3]. Modern reinforcement learning techniques are broadly split into value-based methods, which optimize approximate action-value functions, and policy-gradient techniques, such as Proximal Policy Optimization (PPO), which directly modify the agent’s action probability distribution to maximize performance safely [4, 5]. This thesis leverages these foundational algorithmic families to construct an autonomous control agent capable of navigating space and evading hazards within a custom-designed simulation playground.

1.4 Computer Vision

Computer vision represents a key domain within artificial intelligence dedicated to the mathematical and computational modeling of artificial visual perception, enabling automated machines to autonomously interpret, process, and extract structured semantic properties from digital imagery and real-time video streams [6]. The ultimate ambition of the field is to replicate the intricate, multi-layered perceptual capabilities of the organic visual cortex, transforming unstructured matrices of raw pixel color values into rich, actionable geometric data and high-level situational descriptions [6]. The operational landscape of computer vision encompasses a highly diverse spectrum of specialized downstream processing tasks, which vary significantly in their analytical and spatial complexity [6]. These include foundational image classification, which maps an entire image matrix to a single categorical label; semantic segmentation, which partitions a scene by assigning a discrete class to every individual pixel coordinate; optical flow analysis, which tracks the precise vector velocity of entities across successive temporal frames; and object detection, which bridges classification and spatial localization by placing bounded regions around distinct entities [2, 6].

Historically, the structural evolution of computer vision has been heavily based on the way features are extracted and represented from raw digital image signals [6].

The early methodologies relied completely on manual processing and rigid mathematical modeling, focusing on color space transformations. Convolutional filtering at the Pixel-level, geometric curve fitting, and hand-crafted feature descriptors such as edge-detection matrices [6]. These localized transformations eventually evolved into classical classification frameworks, including clustering algorithms, structural edge extraction, and the Visual Bag of Words paradigm [6]. While these early systems laid the essential groundwork for understanding structural features and scene geometry, they were highly sensitive to changes in environmental lighting, spatial perspective, and background noise, requiring intense manual tuning and resetting by domain experts [2, 6].

The introduction of Deep Learning radically transformed this workflow, establishing Convolutional Neural Networks (CNNs) as the dominant framework for visual feature extraction [2]. CNN architectures are specifically designed to process grid-structured data like images by applying learned spatial filters across the dimensions of an input tensor, allowing the model to share weights and preserve spatial invariance [2]. This operational mechanic allows the network to automatically construct a hierarchical feature extraction pipeline, starting with primitive edge segments and building up to complex object representations [2]. In modern autonomous systems, this visual processing stack forms a vital component of the control cycle, converting complex graphic data into clean spatial tensors that an intelligent agent can safely navigate [3].

1.5 Object Detection

Object detection represents a critical branch of computer vision that simultaneously solves two distinct spatial sub-problems within a unified computational pass: identifying the structural presence of semantic target entities and determining their precise spatial geometry [6]. Unlike standard image classification architectures, which output a single qualitative category label for a complete, isolated pixel matrix, an object detection framework must produce a dynamic array of localized boundaries paired with class probability fields for an arbitrary number of concurrent targets [7]. This complexity requires the computational model to accurately process spatial variations across diverse object scales, overlapping structures, and severe environmental occlusions [6]. Within autonomous navigation loops, object detection acts as the fundamental translation layer, converting high-dimensional visual matrices into structured coordinates

that enable automated decision-making engines to construct a detailed semantic map of their immediate surroundings [3].

Historically, the structural design of computer vision-based detection systems has evolved through two distinct structural neural paradigms:

- **Two-Stage Deep Detectors:** Pioneered by architectures such as the Regional Convolutional Neural Network (R-CNN) family, these frameworks decouple spatial processing into separate components [6, 7]. The initial stage uses a localization model or a regional proposal network to generate a set of candidate bounding regions likely to contain target entities [7]. The secondary stage takes these isolated regional proposals and runs them through independent downstream convolutional classifiers to verify class membership and refine the exact bounding geometries [7]. While two-stage pipelines consistently deliver exceptional localization accuracy, their multi-stage computational complexity results in high inference latency, making them unfeasible for real-time applications operating over high-frame-rate environments [8].
- **One-Stage Real-Time Detectors:** Introduced to eliminate bottlenecks in regional proposals, single-stage architectures reframed object detection as a single regression problem [8]. Models such as the You Only Look Once (YOLO) framework evaluate the raw input tensor in a single, end-to-end forward pass, mapping raw pixels directly to predicted bounding coordinates and categorical probabilities simultaneously [8]. By passing the entire scene through a unified convolutional stack, single-stage detectors optimize inference speed to achieve exceptional processing velocities, providing the low-latency visual performance required to support active reinforcement learning loops in dynamic gaming environments [3, 8].

Modern state-of-the-art iterations, such as YOLOv8, optimize this single-stage pipeline further by transitioning to anchor-free detection heads [9]. Rather than forcing the neural model to predict bounding box deviations relative to a collection of rigid, pre-defined candidate boxes, anchor-free architectures predict the spatial center coordinates and edge boundaries of targets directly [9]. This architectural update significantly simplifies the loss function computation and minimizes non-maximum suppression overhead, allowing autonomous software systems to capture and process live graphic data streams with minimal latency [9]. Using these single-stage processing paradigms, it becomes mathematically feasible to extract relative coordinate arrays

from visual graphics, forming a reliable spatial state representation for a downstream optimization agent [6, 8, 9].

1.6 Reinforcement Learning in Games

Computer game environments have historically served as the primary reference and experimental sandbox for artificial intelligence research, offering highly structured rule frameworks, clearly defined objective functions, and perfectly isolated testing conditions [1]. Throughout the early history of game AI, developers relied almost exclusively on hand-crafted heuristic state charts, finite state machines, and brute-force adversarial tree search algorithms to orchestrate non-player character behavior [10]. While these classical techniques created formidable opponents in fully observable deterministic board games such as chess, they lacked the capacity for domain generalization and suffered from extreme computational scaling constraints when placed in dynamic, continuous or real-time environments where state spaces cannot be mapped out in advance [1, 10]. The emergence of deep reinforcement learning fundamentally shifted this paradigm, introducing autonomous agents capable of independently discovering complex tactical strategies purely through self-play and repeated environmental trial-and-error [3].

A historic milestone in this domain occurred with the development of the Deep Q-Network (DQN) architecture, which achieved human-level control across a diverse suite of Atari 2600 video games [3]. Rather than receiving internal programmatic game variables or engine-level memory states, the DQN agent was fed the raw, unsegmented graphic pixel array from the screen buffer alongside a single scalar reward signal representing the changing game score [3]. This achievement successfully demonstrated that a unified deep learning network could simultaneously solve high-dimensional visual feature extraction and sequential decision-making control loops under a single backpropagation pipeline [3]. Shortly thereafter, this framework was extended to highly intuitive multi-agent domains, combining deep policy gradient formulations with tree search architectures to defeat world-champion human players in the complex board game Go [10]. This series of breakthroughs established deep reinforcement learning as a universally viable methodology for solving real-time navigation, resource distribution, and strategic planning problems [3, 10].

In the context of this thesis, these established game AI principles are mapped directly onto a custom-modeled 2D navigation simulation. Instead of evaluating

pre-defined spatial graphs, the agent must interact with a dynamic plane where it is challenged to maximize resource harvesting efficiency while maintaining survival guarantees against hostile entities. By analyzing how modern value-based and policy-gradient algorithms adapt to this environment, this research builds directly upon the foundational milestones established by modern deep reinforcement learning literature [1, 3, 10].

Chapter 2

Related Work

This chapter reviews the existing academic literature and state-of-the-art methodologies positioning this thesis within the broader research landscape. It explores the historical progression of vision-based reinforcement learning in simulation environments, details prior research that incorporates explicit object detection pipelines within control loops, and examines established behavior optimization techniques for autonomous agents operating in video games.

2.1 Vision-Based Reinforcement Learning in Virtual Environments

The paradigm of training autonomous agents to solve complex tasks directly from visual inputs has been a major focus of modern artificial intelligence research. The foundational milestone in vision-based deep reinforcement learning was achieved by Mnih et al., who successfully trained a Deep Q-Network (DQN) to play a variety of Atari 2600 video games using raw screen pixels as the sole observation state [3]. This work proved that non-linear function approximators could extract abstract representations from raw spatial data and optimize action-value functions simultaneously. However, it also exposed severe limitations regarding sample complexity, requiring tens of millions of environment steps to stabilize basic game-playing policies.

To push the boundaries of vision-based agents into more complex three-dimensional environments, researchers introduced specialized benchmark platforms:

- **VizDoom:** Based on the classic first-person shooter game Doom, this platform forced agents to make decisions in semi-realistic 3D environments from a first-

person perspective, introducing challenges such as visual partial observability, dynamic enemy movement, and delayed rewards [17].

- **MineRL / Minecraft:** Built upon the open-ended sandbox environment of Minecraft, this framework introduced complex hierarchical tasks (such as resource harvesting and tool crafting) where agents must navigate vast, procedurally generated terrains using raw visual feeds and sparse reward structures [18].

Despite the success of these architectures, purely pixel-to-action reinforcement learning suffers from a significant double-learning bottleneck. As highlighted in studies evaluating sample efficiency in deep RL, the internal parameters of a unified end-to-end model must simultaneously learn to solve high-frequency vision tasks (e.g., isolating a moving entity from a shifting background texture) and discover optimal sequential policies [3, 4]. This lack of structural abstraction often causes high gradient variance, training instability, and a tendency to overfit to specific visual conditions, such as minor variations in background lighting or texture colors.

2.2 Object Detection Assisted Reinforcement Learning

To overcome the double-learning bottleneck inherent in raw pixel-to-action networks, researchers have increasingly explored hybrid architectures that combine explicit computer vision extraction models with reinforcement learning backbones. This paradigm, known as Object-Centric Reinforcement Learning (OCRL), attempts to introduce a strong structural inductive bias by forcing the agent to organize its perceptual field around persistent entities rather than raw pixel intensities [19].

Early approaches in this domain relied heavily on unsupervised object discovery frameworks, such as Slot Attention or mask-based segmentation layers, to isolate object representations from video frames without manual annotations [20]. While theoretically elegant, these unsupervised methods often introduce significant architectural complexity, struggle to scale to visually diverse or chaotic environments, and are highly prone to training instability. Consequently, recent literature has shifted toward utilizing pre-trained, high-speed single-stage object detectors to build explicit, structured state abstractions. For instance, in robotic manipulation tasks involving

dynamic environments, hybrid frameworks have successfully deployed YOLO architectures to extract real-time 2D spatial coordinates of moving objects, routing these raw bounding parameters directly into continuous control reinforcement learning heads to guide precise grasping actions [21].

Furthermore, recent investigations into the robustness of deep reinforcement learning models have highlighted the role that structured spatial abstractions play in preventing "shortcut learning"—a phenomenon where pixel-based agents exploit spurious visual correlations, such as background artifacts or localized color transitions, that are completely unrelated to the actual task logic [19]. By integrating a dedicated object detection layer (such as a YOLOv8 network) directly into the agent’s sensory input channel, the environment is filtered into a clean, low-dimensional coordinate vector. Recent benchmarks comparing object-masked visual feeds against raw pixel matrices demonstrate that this structural isolation dramatically reduces sample complexity and enhances the agent’s policy generalization when subjected to novel visual perturbations or environmental layout alterations [22].

2.3 Autonomous Navigation and Behavior Optimization in Games

Autonomous navigation and high-level behavioral control in video games have historically relied on hand-crafted, deterministic pathfinding systems and finite-state machines. In commercial game development, the standard approach to agent movement involves calculating short-to-medium range paths over a pre-compiled navigation mesh (NavMesh) using graph search techniques like the A^* algorithm [23]. While highly efficient for static geometries, these traditional architectures struggle to adapt when environments become dynamic, partially observable, or require complex multi-objective optimization, such as simultaneously hunting targets while avoiding moving hazards under strict time constraints [23].

To address these limitations, deep reinforcement learning has been increasingly adopted to model spatial navigation as a continuous behavior optimization problem. Researchers from major gaming studios have demonstrated that agents trained via deep policy gradients (such as PPO) can learn to navigate complex, vertical 3D video game layouts using generalized movement mechanics, bypassing the need for manual NavMesh link annotations [23]. Furthermore, designing successful navigation and

survival behaviors in these custom virtual spaces relies heavily on shaping the underlying reward function to balance competing operational objectives. Research into autonomous vehicle and mobile robot navigation shows that introducing explicit structural penalty terms—such as continuous step penalties, proximity hazard costs, and clear goal-destination bonuses—prevents the agent from developing passive, stagnant policies, forcing it to discover shorter trajectories and smoother path configurations [24, 25].

In addition to pure geographic traversal, contemporary literature emphasizes the importance of contextual decision-making within survival game loops. When an agent operates in an environment populated by multiple moving entities (e.g., resource nodes, enemy combatants, and walls), its strategy must dynamically prioritize targets based on proximity and immediate survival risk [23, 24]. This thesis builds directly upon these behavioral optimization frameworks; by extracting clean, decoupled coordinate states through a single-stage object detector, the downstream policy network can evaluate spatial navigation vectors natively, maximizing multi-objective rewards without the policy degradation typically caused by visual noise or architectural complexity.

2.3.1 Positioning of This Work

Prior work falls into three categories: (1) pixel-to-action RL [3], which requires millions of samples; (2) object-centric RL using unsupervised discovery [20], which is unstable; (3) hybrid approaches using pre-trained detectors [21]. This thesis builds on approach (3) by demonstrating that YOLOv8-based perception coupled with DQN/PPO achieves rapid convergence in an open-world MMORPG setting.

Chapter 3

Deep Learning Foundations and Neural Architectures

This chapter details the foundational mathematical frameworks, architectural configurations, and optimization paradigms that underpin deep learning systems. By tracking the evolution from organic neurology to multi-layered artificial networks, this structural dive establishes the mechanics behind feedforward processing, error back-propagation, and spatial feature extraction. These mathematical structures form the core components of the vision pipelines and non-linear policy approximators utilized throughout this research.

3.1 Biological Inspiration

The structural design of modern artificial neural networks stems from an effort to mathematically abstract the data-processing capabilities of organic nervous systems [11]. In biological organisms, the fundamental computational block is the neuron, a specialized cell engineered to process and transmit electrochemical signals across complex, organic network structures [12].

The anatomical architecture of a biological neuron is composed of three primary structural sub-components:

- **Dendrites:** A dense branching tree structure that extends from the cell body and acts as the primary input gateway, receiving chemical signals transmitted through structural gaps from neighboring cellular units [12].
- **Soma (Cell Body):** The central processing hub containing the cellular nucleus.

The soma aggregates the incoming electrochemical potentials collected by the dendrites, continuously accumulating the total voltage variation across the cell membrane [11, 12].

- **Axon:** A long, specialized cellular cable extending away from the soma. If the accumulated electrical potential within the soma exceeds a specific physiological threshold, the cell boundary depolarizes, triggering an electrical impulse called an action potential that travels down the length of the axon [12].

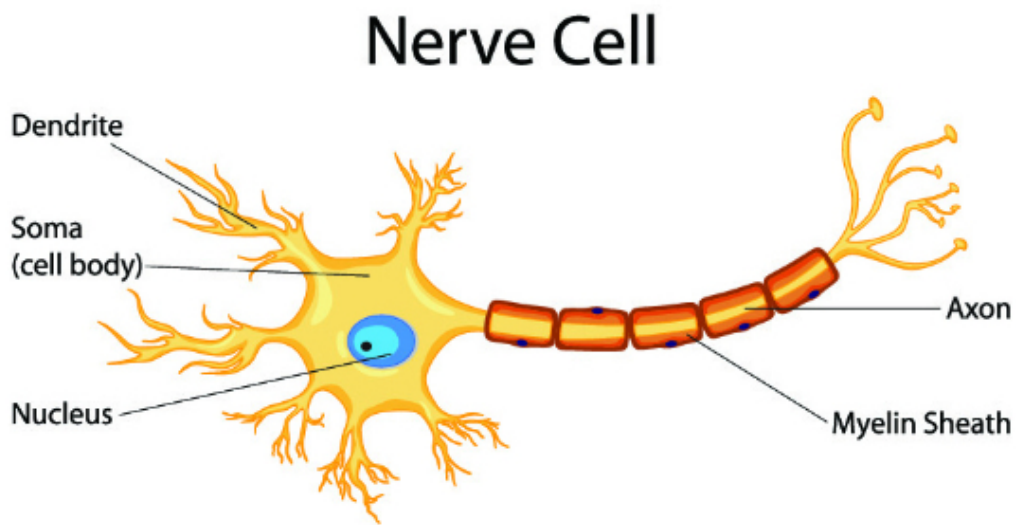


Figure 3.1: The components of a typical neurone.

The transmission interface between the terminal ends of an axon and the receptive dendrites of downstream neurons is regulated by specialized microscopic junctions called synapses [11]. When an action potential reaches a synapse, it triggers the release of chemical neurotransmitters that cross the synaptic gap to bind with receptors on the receiving cell, altering its local electrical potential [12]. The regulatory efficiency of these channels is highly dynamic; synapses can reinforce or inhibit signal strength based on the frequency of historical activation, a physiological property known as synaptic plasticity [11]. This biochemical adaptation forms the biological basis for biological learning, providing the inspiration for the adjustable scalar weights used to train artificial networks [2, 11].

3.2 Artificial Neuron

The artificial neuron, or computational node, is a mathematical abstraction designed to model the signal integration and activation mechanics observed in biological nervous systems [2]. Just as an organic neuron aggregates chemical inputs and generates an output pulse based on an internal threshold, the artificial counterpart maps numerical input vectors into a single scalar output variable through a sequence of linear and non-linear transformations [2].

Mathematically, let an artificial neuron receive a vector of inputs

$$\mathbf{x} = [x_1, x_2, \dots, x_n]^T \in R^n,$$

which represent the incoming signals or raw data features routed into the node. Each input channel is parameterized by a corresponding scalar weight w_i , forming a parameter vector $\mathbf{w} = [w_1, w_2, \dots, w_n]^T \in R^n$ that determines the relative signal importance or connection strength of that specific channel [2]. The linear aggregation stage of the node computes the dot product of these vectors and integrates an explicit additive scalar bias term $b \in R$, which shifts the total activation threshold independent of the input values [2]. This structural sequence yields a single intermediate scalar net input value z :

$$z = \sum_{i=1}^n w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

To replicate the threshold-dependent firing behavior of biological cells, this linear aggregate z is routed into a non-linear activation function $\sigma(\cdot)$ [2]. The activation function transforms the linear output into a bounded or scaled activation state, yielding the final computational output $a = \sigma(z)$. By embedding this mathematical node within dense parallel layers, artificial networks gain the capacity to model multi-dimensional transformations, forming the building blocks of deep feature extraction networks and policy models [2].

3.3 Perceptron

The Perceptron, formulated by Frank Rosenblatt in 1958, represents the historical foundational baseline of supervised binary classification algorithms and serves as the earliest structural implementation of a programmable artificial neuron [2]. Designed as a linear baseline classification model, the classic single-layer Perceptron maps an

incoming numerical input feature vector $\mathbf{x} \in R^n$ directly to a discrete binary output indicator $y \in \{0, 1\}$ by executing a weighted dot-product combination and routing the resulting linear scalar sum through a discontinuous step function [2]. The core mathematical operator defining the Perceptron’s mapping behavior is the Heaviside step activation function, denoted as $f(z)$, which forces the system to produce absolute binary output decisions based purely on the algebraic sign of the aggregated internal potential [13]:

$$y = f(\mathbf{w}^T \mathbf{x} + b) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} + b \geq 0 \\ 0 & \text{if } \mathbf{w}^T \mathbf{x} + b < 0 \end{cases}$$

where $\mathbf{w}$ denotes the adjustable structural weight vector and b represents the baseline scalar bias parameter [13]. Geometrically, the equation $\mathbf{w}^T \mathbf{x} + b = 0$ delineates an $(n - 1)$ -dimensional linear boundary hyperplane that physically bisects the continuous feature input space into two distinct decision half-spaces [2].

The parameter optimization process of Rosenblatt’s architecture is governed by the Perceptron Learning Rule, an iterative error-correction training procedure that updates parameters exclusively when an input pattern is misclassified by the existing boundary hyperplane [2]. Given a training sample pair $(\mathbf{x}_t, y_{\text{true}})$, where $y_{\text{true}} \in \{0, 1\}$, and the model’s current predicted label is denoted as $\hat{y}_t$, the network adjusts its structural parameters according to the following iterative adjustment vectors [13]:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \eta(y_{\text{true}} - \hat{y}_t)\mathbf{x}_t \\ b_{t+1} &= b_t + \eta(y_{\text{true}} - \hat{y}_t) \end{aligned}$$

where $\eta \in (0, 1]$ represents a constant scalar parameter known as the learning rate [13]. If the training sample is classified correctly, the difference term $(y_{\text{true}} - \hat{y}_t)$ drops to zero, and the network parameters remain entirely unchanged. The Perceptron Convergence Theorem guarantees that if the underlying dataset is strictly linearly separable—meaning there exists a spatial hyperplane configuration that can isolate the two target classes without error—the learning rule will converge to a valid separating boundary in a finite number of training steps [2, 13].

However, this architecture suffers from a catastrophic structural limitation, famously demonstrated by Marvin Minsky and Seymour Papert in 1969: a single-layer Perceptron is mathematically incapable of solving non-linearly separable operations, such as the fundamental exclusive-or (XOR) logic gate configuration [2]. Because the

XOR gate requires a non-linear decision boundary to separate its categorical outputs in a two-dimensional coordinate system, the single linear hyperplane of the Perceptron fails completely, a structural limitation that historically halted neural network research until the development of multi-layered architectures [2].

3.4 Multi-layer Perceptron

The structural limitations of single-layer models led directly to the development of the Multi-layer Perceptron (MLP), a feedforward artificial neural network architecture composed of a deeply stacked hierarchical network of computational nodes [2]. By introducing one or more intermediate computational structures, termed hidden layers, between the input interface and the terminal output layer, the MLP circumvents the geometric constraints of linear hyperplanes [14]. This structural expansion transforms the model into a universal function approximator, capable of mapping highly complex, non-linear continuous target functions across high-dimensional coordinate spaces [2, 14].

The architectural topology of a standard Multi-layer Perceptron is organized as a directed feedforward graph where each layer is fully connected to the subsequent layer, meaning every individual node in layer l distributes its calculated activation value to every single neuron residing in the downstream layer $l + 1$ [14]. Crucially, there are no structural lateral connections between nodes within the same layer, and no recurrent feedback loops mapping data backward during the inference cycle [2]. Mathematically, let $\mathbf{a}^{[l]}$ represent the vector of activation values produced by layer l , where $\mathbf{a}^{[0]} = \mathbf{x}$ represents the raw input tensor injected into the system [14]. The forward processing transformation for any subsequent hidden layer l within the architecture is parameterized by a dedicated weight matrix $\mathbf{W}^{[l]}$ and an additive bias vector $\mathbf{b}^{[l]}$, computed sequentially as follows [2]:

$$\begin{aligned}\mathbf{z}^{[l]} &= \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{a}^{[l]} &= \sigma(\mathbf{z}^{[l]})\end{aligned}$$

where $\mathbf{z}^{[l]}$ represents the intermediate vector of linear net inputs for layer l , and $\sigma(\cdot)$ denotes an element-wise non-linear activation function [2]. By substituting the discontinuous step functions of the classical Perceptron with continuous, differentiable non-linear activation functions, the Multi-layer Perceptron gains the ability to deform

and project input feature spaces into highly complex coordinate structures [14]. This hierarchical transformation allows the network to automatically extract complex latent features and discover highly non-linear decision boundaries, making the MLP exceptionally suited to serve as the non-linear policy core and value-function approximator within modern deep reinforcement learning pipelines [3, 14].

3.5 Forward Propagation

Forward propagation represents the sequential mathematical execution pipeline through which an artificial neural network processes an input tensor to generate a corresponding output prediction or behavioral hypothesis [2]. The structural operation flows strictly in a forward direction through the layered graph topology, moving from the initial input layer, through every intermediate hidden layer, to the final output layer, without containing any backward recurrent feedback channels [14].

To formalize this computational processing cycle, let the input tensor injected into the network be designated as $\mathbf{a}^{[0]} = \mathbf{x} \in R^n$ [14]. For any given layer l within an L -layer multi-layer perceptron, the network must aggregate the outputs from the preceding layer $l - 1$ by applying an explicit linear transformation matrix [2]. This linear stage is parameterized by a layer-specific weight matrix $\mathbf{W}^{[l]}$ and an additive layer-specific scalar bias vector $\mathbf{b}^{[l]}$ [14]. The intermediate linear pre-activation tensor, denoted as $\mathbf{z}^{[l]}$, is calculated through a matrix dot-product compilation [2]:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

Following this linear integration, the pre-activation vector $\mathbf{z}^{[l]}$ is passed directly through an element-wise, non-linear activation function $\sigma(\cdot)$ to produce the final activated output tensor of that layer, designated as $\mathbf{a}^{[l]}$ [2]:

$$\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]})$$

This two-stage transformation is repeated for every layer $l \in \{1, 2, \dots, L\}$ across the structural depth of the network [14]. At the terminal layer L , the network evaluates the raw activated values $\mathbf{a}^{[L]}$ to output the model’s final prediction, which represents estimated action-utility values $Q(s, a)$ or action selection probabilities $\pi(a|s)$ depending on the architectural choice of the reinforcement learning agent [1, 3].

3.6 Backpropagation

The backpropagation algorithm, formalized by Rumelhart, Hinton, and Williams in 1986, is the mathematical engine that enables the optimization of deep neural networks by computing the exact partial derivatives of a scalar loss function with respect to every weight and bias parameter in the system [2]. Rather than recalculating gradients independently for each individual parameter, backpropagation leverages the mathematical chain rule to efficiently propagate error metrics backward from the terminal output layer down to the initial input interface, drastically reducing computational overhead [14].

The execution sequence begins after forward propagation has produced a final prediction $\mathbf{a}^{[L]}$ and a scalar loss function value $\mathcal{L}$ has been calculated against a known ground-truth target vector [2]. The algorithm first evaluates the error gradient at the terminal output layer L , computing how changes in the pre-activation tensor $\mathbf{z}^{[L]}$ affect the global loss value [14]. This localized output layer error tensor, denoted as $\delta^{[L]}$, is calculated as the element-wise product of the loss gradient and the first derivative of the output activation function [2]:

$$\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[L]}} \odot \sigma'(\mathbf{z}^{[L]})$$

Once the terminal error vector $\delta^{[L]}$ is established, the algorithm uses the mathematical chain rule to propagate this error backward through each hidden layer $l = L - 1, L - 2, \dots, 1$ [14]. The localized error tensor $\delta^{[l]}$ for any internal hidden layer is determined by projecting the error from the subsequent downstream layer back through the transposed weight matrix $\mathbf{W}^{[l+1]}$ and multiplying by the local activation derivative [2]:

$$\delta^{[l]} = ((\mathbf{W}^{[l+1]})^T \delta^{[l+1]}) \odot \sigma'(\mathbf{z}^{[l]})$$

Using these localized layer error vectors, the exact partial derivatives of the loss function with respect to the individual weight matrices and bias vectors can be computed directly from local data parameters [14]. Specifically, the gradient of the loss with respect to a weight matrix is the outer product of that layer's error vector and the incoming activation values from the preceding layer, while the bias gradient maps directly to the error vector itself [2]:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} (\mathbf{a}^{[l-1]})^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

These calculated gradient tensors represent the exact directional vectors of steepest error ascent across the loss surface [2]. They are passed directly to downstream optimization algorithms to iteratively adjust the network’s parameters, reducing prediction errors over successive training runs [14].

3.7 Activation Functions

Activation functions are fundamental mathematical operators injected into the hidden and output layers of an artificial neural network to introduce non-linearity into the system’s processing pipeline [2]. Without the integration of non-linear activation functions, any multi-layered feedforward network would mathematically collapse into a single linear regression model, as any sequential combination of linear matrix transformations remains strictly a linear transformation [14]. Consequently, these operators enable deep architectures to deform data spaces and map highly intricate, non-linear relationships between inputs and outputs [2].

Historically and architecturally, several activation functions are utilized to govern the firing behavior of neural nodes:

- **Sigmoid Function:** Mathematically defined as $\sigma(z) = \frac{1}{1+e^{-z}}$, this function maps any continuous real-valued scalar into a strictly bounded open interval between 0 and 1 [2]. While highly useful for interpreting outputs as probabilities, the Sigmoid function suffers from severe saturation regions; when the input values are exceptionally large or small, the derivative of the function approaches zero ($\sigma'(z) \rightarrow 0$) [2]. This structural behavior causes the vanishing gradient problem during backpropagation, effectively halting parameter updates in deep networks [14].
- **Hyperbolic Tangent (tanh) Function:** Formulated as $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, this function maps inputs into a symmetric range between -1 and 1 [14]. Because it is zero-centered, the regularized output values have a mean close to zero, which typically improves the optimization convergence speed compared to

the Sigmoid function [2]. However, it remains highly susceptible to the same gradient saturation and vanishing problems at extreme input thresholds [2].

- **Rectified Linear Unit (ReLU):** Defined by the piecewise operator $f(z) = \max(0, z)$, ReLU has become the standard activation function for hidden layers in modern deep learning models [2]. Because its derivative is a constant 1 for all positive inputs ($z > 0$), it does not saturate in the positive domain, significantly mitigating the vanishing gradient problem and accelerating model training times [2]. Despite its advantages, it can suffer from the "dying ReLU" problem, where nodes that receive strictly negative inputs across the dataset output a constant zero with a zero gradient, rendering them permanently inactive during optimization [2].

3.8 Loss Functions

A loss function, or cost function, is a formalized mathematical objective function that maps the discrepancy between a neural network's empirical predictions and the corresponding ground-truth target labels into a single scalar error metric [2]. This metric quantifies the operational performance of the network, acting as the explicit signal that optimization algorithms minimize during backpropagation [14]. The choice of a loss function is strictly dictated by the mathematical nature of the learning task being solved [2].

For regression paradigms, where the model is tasked with predicting continuous scalar values, the standard metrics are Mean Squared Error (MSE) and Mean Absolute Error (MAE) [2]. Mean Squared Error computes the average of the squared differences between predicted values $\hat{y}_i$ and true values y_i , heavily penalizing large outlier errors due to its quadratic formulation [2]:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

For categorical classification tasks, the framework relies on Cross-Entropy Loss, which derives from information theory principles and measures the distance between two probability distributions [2]. For a binary classification setup, Binary Cross-Entropy evaluates the probability output $\hat{y} \in [0, 1]$ against the actual binary class $y \in \{0, 1\}$ [2]:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

When scaled to multi-class classification tasks or object detection bounding box classification heads, this generalizes to Categorical Cross-Entropy, combined with a Softmax output layer to evaluate categorical distributions across multiple distinct target categories [2]. In reinforcement learning loops, specialized variants are used, such as Mean Squared Bellman Error for value estimation or clipped surrogate objectives to secure stable policy adjustments [1, 4].

3.9 Optimization (Gradient Descent, Adam)

Optimization algorithms are the mathematical frameworks used to update the internal weights and biases of a neural network to minimize the scalar output of a chosen loss function [2]. These algorithms use the gradient vector calculated during backpropagation, which points in the direction of steepest error ascent across the multi-dimensional parameter landscape, to iteratively adjust parameters in the opposite direction toward a global or local minimum [14].

The fundamental baseline optimization method is Stochastic Gradient Descent (SGD) [2]. In its standard form, SGD updates the network parameter vector θ by calculating the gradient of the loss function $\nabla_{\theta} \mathcal{L}$ over a small subset or mini-batch of training samples [2]:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}(\theta_t)$$

where $\eta > 0$ represents the learning rate, a hyperparameter that controls the step size taken down the error gradient [2]. While mathematically straightforward, standard SGD can suffer from slow convergence in regions where the loss surface is highly un-isotropic or forms ravines, and choosing an optimal static learning rate remains a difficult challenge [2]. To accelerate optimization, momentum methods are often added, which introduce a velocity vector that accumulates past gradients to dampen oscillations and speed up descent along consistent directional paths [2, 14].

Modern deep learning architectures regularly rely on more sophisticated adaptive learning rate algorithms, most notably Adaptive Moment Estimation (Adam) [15]. The Adam optimizer computes individual adaptive learning rates for different parameters by keeping tracking estimates of both the first raw moment (the mean) and the

second uncentered moment (the uncentered variance) of the incoming gradients [15]. The algorithm maintains an exponentially decaying average of past gradients m_t and past squared gradients v_t , parameterized by decay hyperparameters $\beta_1, \beta_2 \in [0, 1)$ [15]:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

where $g_t = \nabla_{\theta} \mathcal{L}(\theta_t)$ represents the gradient at time step t [15]. Because m_t and v_t are typically initialized as zero vectors, they are biased toward zero, especially during the initial training steps [15]. To correct this, Adam applies bias-correction terms to produce rehabilitated moment estimates [15]:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned}$$

The final parameter update formula divides the corrected first moment by the square root of the corrected second moment, ensuring that parameters with highly volatile gradients receive smaller, more stable updates, while parameters with sparse gradients take larger steps [15]:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

where $\epsilon > 0$ is a tiny smoothing term used to prevent division-by-zero errors [15]. This adaptive capability makes Adam highly robust across varying network architectures and complex loss surfaces, making it a standard choice for training deep reinforcement learning models and real-time object detectors [3, 9].

3.10 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized class of deep neural network architectures designed to process grid-structured data matrices, such as digital images or video frame buffers [2]. While standard fully connected layers scale poorly when applied to high-dimensional visual data—requiring an independent weight parameter for every single pixel-to-neuron connection—CNNs exploit spatial properties

by enforcing three architectural principles: localized receptive fields, shared weight matrices, and spatial downsampling [2].

The foundational core of a CNN is the convolutional layer, which replaces standard matrix multiplication with a localized convolutional operation [2]. Instead of processing the entire input tensor simultaneously, the layer slides a small, learnable three-dimensional parameter matrix, known as a kernel or filter, across the spatial width and height of the input data [2]. At each localized position, the network computes a scalar dot product between the kernel weights and the small overlapping patch of input pixels, summing the values along with a bias term to produce a single output coordinate in a structured feature map [14]. As the filter systematically sweeps the input according to a defined stride value, it constructs a dense spatial representation of localized features [2]. Because the same kernel weight matrix is applied across the entire input grid, the network achieves weight sharing, which drastically reduces the total parameter count and ensures translation invariance—meaning a visual feature like an edge or a boundary can be detected regardless of its exact coordinate position within the image frame [2].

To manage computational complexity and build a hierarchy of features, convolutional layers are typically paired with pooling layers [2]. The pooling layers perform a non-linear downsampling operation across the spatial dimensions of the feature maps, reducing their width and height while preserving the most prominent activated values [14]. The most common implementation is Max Pooling, which slides a small window across the feature map and retains only the maximum value within that region, discarding redundant spatial noise and increasing the network’s abstract field of view [2].

Through successive layers of convolution, non-linear activation, and spatial pooling, a CNN automatically constructs a deep feature hierarchy [2]. The early layers isolate primitive structural components like edges and color contrasts, intermediate layers combine these segments into geometric shapes, and final fully connected layers map these abstract patterns to high-level semantic outputs [2]. In real-time computer vision tasks, this structural pipeline forms the backbone of modern single-stage object detection architectures like YOLO, extracting the clean spatial feature maps required to locate and track target objects instantly [8, 9].

Chapter 4

Reinforcement Learning Frameworks and Policy Optimization

This chapter establishes the formal mathematical framework of reinforcement learning used to develop autonomous decision-making agents. Beginning with the foundational optimization mechanics of Markov Decision Processes, the text details the theoretical evolution from classical tabular solutions to modern deep value-based and policy-gradient paradigms. These algorithms enable an agent to learn optimal navigation and survival strategies directly from dynamic environmental feedback.

4.1 Markov Decision Process

The formal structural foundation of almost all modern reinforcement learning environments is mathematically framed as a Markov Decision Process (MDP) [1]. An MDP provides a precise algebraic specification for modeling sequential decision-making loops where outcomes are partly under the control of an autonomous agent and partly influenced by stochastic transitions within the environment [14].

The underlying bedrock of this framework is the strict enforcement of the Markov Property [1]. A stochastic transition process is said to possess the Markov property if the conditional probability distribution of all future states depends exclusively upon the immediate current state and selected action, remaining entirely independent of the historical path, past states, or prior actions taken by the agent [1]. Mathematically, this memoryless property is defined as [1]:

$$\begin{aligned}
P(S_{t+1} = s', R_{t+1} = r \mid S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \dots, S_0 = s_0) = \\
= P(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)
\end{aligned}$$

A formal Markov Decision Process is structurally defined as a 5-tuple tuple block $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, which encapsulates the complete operational physics of the simulation environment [1]:

- $\mathcal{S}$ represents the State Space, a comprehensive set containing all valid environmental configurations [1].
- $\mathcal{A}$ represents the Action Space, a comprehensive set containing all valid moves or behavioral choices available to the agent [1].
- $\mathcal{P}$ represents the dynamic Transition Probability Function, defined as $\mathcal{P}(s' \mid s, a) = P(S_{t+1} = s' \mid S_t = s, A_t = a)$, mapping the probability of landing in state s' after taking action a in state s [14].
- $\mathcal{R}$ represents the Reward Function, defined as $\mathcal{R}(s, a, s') = E[R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s']$, returning a scalar feedback value based on transitions [1].
- $\gamma \in [0, 1]$ represents the scalar Discount Factor, which determines the relative importance of immediate rewards versus long-term strategic payoffs [1].

4.1.1 State

The state $s \in \mathcal{S}$ represents a formal numerical encapsulation of the environment's current physical configuration at any specific discrete time step t [1]. In an ideal, fully observable MDP, the state vector provides a complete, self-contained description of the environment, containing all the information necessary for the agent to compute an optimal action choice without requiring historical data records [14]. In practical game-based applications, the state representation must be engineered to balance completeness with computational tractability [3]. For the navigation task explored in this thesis, the state space can be structured either as a continuous matrix of raw pixels extracted from the screen buffer or as a dense, processed array of spatial vectors containing the exact coordinates and relative distances of target objectives and hostile entities [3, 6].

4.1.2 Action

The action $a \in \mathcal{A}$ represents the discrete behavioral choice or control command executed by the learning agent at time step t , triggering an environmental state transition [1]. Depending on the design of the environment, the action space $\mathcal{A}$ is mathematically classified into two structural types [14]:

- **Discrete Action Space:** The agent chooses from a finite set of distinct, pre-defined movements or commands [1]. For a 2D game environment, this typically maps to a localized directional array, such as $\mathcal{A} = \{\text{Up, Down, Left, Right, Idle}\}$.
- **Continuous Action Space:** The agent outputs real-valued scalar vectors belonging to a continuous range, allowing for infinitely fine adjustments [4]. This scales directly to physical navigation, controlling precise steering angles or explicit thrust velocities.

4.1.3 Reward

The reward $r_t \in R$ is a scalar feedback signal returned by the environment immediately following an agent's action transition [1]. The reward function acts as the formal translation of the designer's goal into a numerical optimization metric, directly guiding the policy discovery process [14]. The primary directive of the agent is to maximize the expected cumulative return, meaning the reward structure must be calibrated to avoid unintended behavioral side effects [1]. In a navigation and survival scenario, positive rewards are awarded for harvesting target resources, while negative penalties are injected when the agent collides with hazardous obstacles or boundaries [3].

4.1.4 Policy

The policy, denoted as π , represents the internal strategy or decision-making function that governs the agent's behavioral choices across the state space [1]. Formally, a policy maps an input state vector directly to an output action selection distribution [14]. Policies are mathematically categorized into two procedural types [1]:

- **Deterministic Policy:** A direct mapping function $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$ that always assigns a single, specific action to a given state without any stochastic variance [14].

- **Stochastic Policy:** A probability distribution mapping $\pi(a \mid s) = P(A_t = a \mid S_t = s)$, which outputs the conditional probability of executing action a when encountering state s [1]. Stochastic policies are vital for balancing exploration with exploitation, allowing the agent to discover novel trajectories in uncertain environments [4].

4.1.5 Value Function

Value functions are mathematical estimators that quantify the long-term expected cumulative utility of a state, enabling an agent to evaluate the strategic worth of its current position beyond immediate reward feedback [1]. The State-Value Function, denoted as $V^\pi(s)$, estimates the expected total discounted return achieved when starting from state s and strictly following policy π thereafter [14]:

$$V^\pi(s) = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$$

To evaluate specific physical actions independently of the active policy configuration, algorithms utilize the Action-Value Function, or Q-function, denoted as $Q^\pi(s, a)$ [1]. The Q-function computes the expected long-term discounted return achieved by executing action a within state s , subsequently adhering to policy π for all future operational steps [5]:

$$Q^\pi(s, a) = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]$$

These value architectures form the foundational core of reinforcement learning optimization, allowing agents to systematically rank behavioral alternatives and update their decision models based on empirical returns [1, 5].

4.2 Bellman Equation

The Bellman Equation, formulated by Richard Bellman in the 1950s, establishes a fundamental recursive relationship that breaks down a value function into an immediate reward signal and the discounted value of the subsequent downstream state [1]. This mathematical formulation underpins the computational mechanics of reinforcement learning, transforming long-term value estimation from an intractable infinite-horizon summation into an iterative step-by-step update rule [14].

For any arbitrary behavioral strategy π , the Bellman Expectation Equation for the action-value function expresses the utility of a state-action pair (s, a) as the immediate reward plus the expected discounted value of the next state-action pair (s', a') , averaged across the combined transition physics of the environment and the action probabilities of the policy [1]:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) \left[\mathcal{R}(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a' | s') Q^\pi(s', a') \right]$$

To find the absolute best control strategy, reinforcement learning focuses on discovering the optimal policy π^* , which maximizes the expected return across all states [14]. The maximum utility profile achievable across the environment is defined by the Bellman Optimality Equation for the action-value function [1]. This formulation asserts that the optimal value of a state-action pair must match the immediate reward plus the maximum discounted return achievable by the single best downstream action choice, completely independent of the current policy's distribution [1, 5]:

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) \left[\mathcal{R}(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a') \right]$$

The Bellman Optimality Equation is a non-linear system of equations [1]. While dynamic programming methods can solve this system exactly when the environmental transitions $\mathcal{P}$ and rewards $\mathcal{R}$ are fully known, model-free reinforcement learning algorithms are designed to approximate this optimal surface through direct interaction with the environment, bypassing the need for explicit transition models [1, 5].

4.3 Q-learning

Q-learning, introduced by Chris Watkins in 1989, is an off-policy temporal-difference learning algorithm designed to solve the Bellman Optimality Equation through experience without requiring a model of the environment's transition dynamics [5]. The term "off-policy" indicates that the algorithm estimates the optimal action-value function Q^* independently of the behavioral policy used to select exploratory actions during training [1].

In its classical implementation, Q-learning maintains an explicit, tabular memory matrix known as a Q-table, which allocates a dedicated scalar slot for every unique state-action combination $(s, a) \in \mathcal{S} \times \mathcal{A}$ [5]. The values in this table are initialized to arbitrary baselines and iteratively refined as the agent interacts with the environment

[5]. At each discrete time step t , the agent observes the current state s , executes an action a chosen via an exploratory policy (such as ϵ -greedy), and observes an immediate scalar reward r along with a subsequent destination state s' [1]. The algorithm then updates the corresponding slot in the table using the following temporal-difference formula [5]:

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

where $\alpha \in (0, 1]$ represents the learning rate, controlling the update step size, and γ is the discount factor [1]. The expression inside the brackets represents the Temporal Difference (TD) Error, which measures the discrepancy between the immediate empirical estimate (the TD Target: $r + \gamma \max_{a'} Q(s', a')$) and the current stored value $Q(s, a)$ [1].

The Q-learning convergence theorem guarantees that if the state and action spaces are finite, the learning rate satisfies standard stochastic approximation conditions, and all state-action pairs are visited infinitely often, the tabular values $Q(s, a)$ will converge with probability one to the true optimal action-value function Q^* [5]. However, this tabular approach faces severe scaling limitations when applied to real-world tasks or complex video games [3]. As the number of dimensions in the state space increases, the total number of individual table rows grows exponentially—a phenomenon known as the curse of dimensionality—rendering classical tabular Q-learning computationally intractable for environments with continuous or high-dimensional visual states [1, 3].

4.4 Deep Q Network

The Deep Q-Network (DQN), developed by Mnih et al. in 2015, solved the scaling limits of tabular reinforcement learning by replacing the rigid Q-table with a deep neural network function approximator [3]. Rather than storing explicit values for every individual state-action pair, a DQN architecture passes a high-dimensional state vector $\mathbf{x}$ (such as raw screen pixel frames) through a series of convolutional and fully connected layers, outputting an array of continuous scalar value estimates $Q(s, a; \theta)$ for all available actions simultaneously, where θ represents the weight and bias parameters of the network [3].

Directly combining deep non-linear neural networks with temporal-difference reinforcement learning can lead to severe training instabilities or divergence due to three

main factors: highly correlated successive data samples, non-stationary training targets, and a feedback loop where the network’s current policy strongly dictates its incoming data distribution [3]. To address these stability issues, the DQN framework introduces two core mechanisms [3]:

- **Experience Replay Buffer:** Instead of updating the network parameters immediately using live sequential transitions, each experience tuple

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

is stored in a large rolling memory buffer $\mathcal{D}$ [3]. During the optimization phase, the algorithm samples small, uniformly random mini-batches of transitions from this buffer to calculate the loss gradients [3]. This random sampling breaks the temporal correlation between consecutive states, transforming the incoming data stream into an independent and identically distributed (i.i.d.) structure that satisfies the core assumptions of stochastic gradient descent optimization [2, 3].

- **Target Network Layer Alignment:** To eliminate non-stationary target oscillations, DQN maintains two structurally identical neural networks: a policy network parameterized by weights θ , which is actively updated at every training step, and a separate target network parameterized by weights θ^- , which remains frozen for a fixed number of iterations [3]. The target network is used exclusively to compute the loss function’s downstream evaluation target [3]. The Mean Squared Bellman Error loss function minimized by the active network is defined as [3]:

$$\mathcal{L}(\theta) = E_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right]$$

By decoupling the active update parameters θ from the target calculation parameters θ^- , the optimization landscape behaves like a stationary regression task over short horizons, allowing the system to achieve stable convergence when learning directly from high-dimensional visual feeds [3].

4.5 Double DQN

While standard Deep Q-Networks significantly improve learning stability, they suffer from a systematic architectural flaw known as maximization bias, which can lead to

severe overestimations of action-utility values [16]. This bias stems directly from the use of the mathematical maximization operator $\max_{a'} Q(s', a'; \theta^-)$ in the standard DQN target formula [16]. Because the same target parameter set θ^- is responsible for both choosing the single best downstream action and estimating its corresponding long-term utility value, any random statistical noise or un-modeled error variance in the network’s function approximations will systematically skew the target value upward, causing the algorithm to overvalue sub-optimal trajectories [16].

To eliminate this maximization bias, Hasselt, Guez, and Silver formulated the Double Deep Q-Network (Double DQN) architecture in 2016 [16]. The core principle of Double DQN is to completely decouple the action selection mechanism from the action evaluation step by splitting the roles between the active policy network θ and the frozen target network θ^- [16].

Mathematically, when computing the temporal-difference target for a sampled experience transition, the Double DQN algorithm first queries the active policy network θ to determine which action a^* maximizes the estimated utility in the next state s' [16]:

$$a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s', a; \theta)$$

Once this optimal action is selected, the algorithm routes it to the target network θ^- to evaluate its actual utility score [16]. The updated Double DQN loss function is defined as [16]:

$$\mathcal{L}_{\text{DDQN}}(\theta) = E_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma Q(s', \operatorname{argmax}_{a' \in \mathcal{A}} Q(s', a'; \theta); \theta^-) - Q(s, a; \theta) \right)^2 \right]$$

By utilizing this two-step calculation, an overestimation error in the active network is highly unlikely to match an overestimation error in the target network for the same action, significantly reducing value inflation [16]. Empirically, this modification yields much more stable and accurate value estimations, which directly translates to faster training convergence and more consistent navigation performance for autonomous agents operating in dynamic gaming environments [16].

4.6 Proximal Policy Optimization

While value-based methods like Deep Q-Networks perform exceptionally well in discrete action environments, they are less suited for continuous control tasks and can

experience instability due to sudden policy shifts [4]. Proximal Policy Optimization (PPO), introduced by Schulman et al. in 2017, is a model-free policy gradient method designed to ensure training stability by preventing the updated policy from deviating too far from the previous policy baseline [4].

PPO operates as an actor-critic architecture, utilizing two distinct neural components: the actor, which maps environmental states to action probability distributions $\pi_{\theta}(a|s)$, and the critic, which estimates the state-value function $V(s)$ to establish a baseline for performance evaluation [4]. The core mechanism that stabilizes optimization is the clipped surrogate objective function [4]. Let the probability ratio between the newly updated policy and the old policy be defined as $r_t(\theta)$ [4]:

$$r_t(\theta) = \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)}$$

Instead of maximizing a standard policy gradient objective, which can cause catastrophically large parameter updates, PPO limits the incentive for the ratio $r_t(\theta)$ to move outside a small interval around 1 [4]. The objective function modifies the update by selecting the minimum of a standard unclipped adjustment and a clipped variation [4]:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$$

where ϵ is a hyperparameter (typically 0.1 or 0.2) and $\hat{A}_t$ is the calculated Generalized Advantage Estimate, which quantifies whether a selected action performed better or worse than the critic’s baseline prediction [4]. Taking the minimum of the clipped and unclipped values creates a pessimistic bound; the algorithm ignores improvements that would require an excessively large policy shift, ensuring stable step-by-step updates even when optimizing complex behavioral patterns [4].

4.7 Comparison of Paradigms

To select the optimal control architecture for a given task, it is necessary to analyze the mathematical trade-offs between value-based and policy-gradient frameworks. The primary operational distinctions are summarized in the comparative overview below [1, 3, 4, 16]:

Table 4.1: Comparative Analysis of Reinforcement Learning Paradigms

Metric	Tabular Learning	Q-	Deep Networks (DQN)	Q-	Proximal Policy Optimization
Action Space	Discrete (Small)		Discrete Only		Discrete or Continuous
State Space	Finite, Dimensional	Low-	Continuous, High-Dimensional		Continuous, High-Dimensional
Memory Layout	Explicit Array	Matrix	Deep Neural Approximator	Ap-	Actor-Critic Dual Networks
Optimization	Temporal Difference Error	Differ-	Mean Squared Bellman Error		Clipped Surrogate Objective
Sample Policy	Off-Policy		Off-Policy		On-Policy
Limitations	Curse of Dimensionality	Dimen-	Overestimation Bias		High Sample Complexity

Tabular Q-learning provides strong mathematical convergence guarantees, but its dependence on explicit matrix tables limits its practical application to simple environments with small state spaces [5]. Deep value approximators like DQN and Double DQN use convolutional and dense hidden layers to solve high-dimensional tasks, allowing agents to navigate directly from complex visual fields or vector spaces [3, 16]. However, they remain architecturally restricted to discrete action sets, preventing their use in applications that require continuous steering or velocity adjustments [4].

Policy gradient methods like PPO solve these limitations by parameterizing the strategy distribution directly, allowing the agent to operate effectively in continuous action environments [4]. While PPO’s on-policy design requires a higher volume of training samples—since data cannot be reused across different policy updates—its clipped objective function delivers exceptional training stability [4]. This makes it highly effective for complex navigation and survival tasks where minor parameter oscillations could otherwise derail the learning process [4].

Chapter 5

Object Detection and Computer Vision Pipelines

This chapter details the structural configuration of the computer vision pipelines used to extract spatial awareness from dynamic pixel environments. By tracking the progression from raw feature extraction up to real-time object detection models, this section establishes the mathematical and architectural frameworks required to transform raw screen streams into actionable state coordinates for an automated decision-making agent.

5.1 Computer Vision Pipeline

A computer vision pipeline represents a structured, multi-stage processing system designed to ingest raw digital imagery, apply pixel-level enhancements, and systematically extract high-level semantic data [6]. Within the control loop of an autonomous agent, this pipeline serves as the primary sensory translation layer, bridging the gap between unstructured color matrices and clean, low-dimensional coordinate spaces [3].

The operational lifecycle of a standard vision pipeline is systematically broken down into three distinct layers of abstraction:

1. **Low-Level Processing (Preprocessing):** This initial stage interfaces directly with the raw pixel grid to normalize environmental variance and minimize downstream computational overhead [6]. Typical transformations include spatial resizing to match structural network dimensions, color-space mapping (such as transforming three-channel RGB arrays into single-channel grayscale tensors),

and global contrast alignment or pixel value scaling between the intervals $[0, 1]$ or $[-1, 1]$.

2. **Mid-Level Processing (Feature Extraction):** In this stage, the pipeline isolates structural patterns, spatial gradients, and geometric configurations from the preprocessed tensor without yet assigning explicit categorical labels [2]. This is accomplished by sweeping localized mathematical kernels across the image space to detect physical transitions.
3. **High-Level Processing (Task Ingestion):** The final analytical stage where the extracted feature maps are routed into specialized neural output heads [6]. These mathematical heads process the latent structural abstractions to solve dedicated downstream estimation tasks, such as discrete category mapping, geometric localization boundaries, or real-time spatial coordinate tracking.

5.2 Feature Extraction

Feature extraction is the mathematical and computational process of transforming high-dimensional, raw image pixel tensors into a lower-dimensional vector representation that selectively preserves critical structural properties [2]. A raw digital image frame contains an immense volume of redundant or noisy data; processing these raw arrays directly through dense decision-making layers can lead to rapid parameter explosion, overfitting, and severe computational bottlenecks [14].

In modern computer vision systems, feature extraction is handled automatically by stacking deep Convolutional Neural Networks (CNNs) [2]. As an input tensor passes through consecutive convolutional operations, the network builds a structural abstraction hierarchy [2]:

- **Shallow Layers:** Isolate high-frequency local changes, detecting basic edge segments, ridge boundaries, and primitive color contrasts [2].
- **Deep Layers:** Combine these primitive edges through repeated matrix operations and spatial downsampling to capture complex geometric structures, semantic parts, and abstract target representations [6].

This output feature map compresses spatial data while retaining structural context, forming the foundation for downstream classification and localization tasks [14].

5.3 Image Classification

Image classification is the foundational task in computer vision where a neural network maps an entire digital image matrix to a single discrete categorical label y chosen from a pre-defined set of classes $\mathcal{Y}$ [2].

Mathematically, given an input tensor $\mathbf{X}$, a classification network projects the data through a feature extraction backbone to yield a raw logit vector $\mathbf{z}$ [14]. This vector is passed through a Softmax activation layer to produce a normalized probability distribution over all target categories [2]:

$$P(y = k \mid \mathbf{X}) = \frac{e^{z_k}}{\sum_{j=1}^{|\mathcal{Y}|} e^{z_j}}$$

The network is trained by minimizing Categorical Cross-Entropy loss, which penalizes discrepancies between the predicted probabilities and true one-hot encoded vectors [2]. While image classification provides global semantic understanding (answering *what* is contained within a scene), it lacks spatial specificity; the model assumes a single dominant target and provides no information regarding its position, scale, or quantity within the broader environment [6].

5.4 Localization

Localization extends the classification paradigm by identifying the exact spatial coordinates of a target object within the image frame, combining qualitative categorization with quantitative spatial mapping [6].

To achieve this, the network architecture splits into dual output heads after the feature extraction backbone [7]:

1. **Classification Head:** Predicts the class label probability distribution using a Softmax layer [2].
2. **Regression Head:** Outputs a four-dimensional continuous scalar vector $\mathbf{b} = [x, y, w, h]^T \in R^4$ representing the target’s bounding box spatial center coordinates (x, y) , width w , and height h , normalized relative to the image boundaries [7].

The optimization of the regression head relies on continuous distance metrics, such as the Mean Squared Error (L_2 Loss) or Mean Absolute Error (L_1 Loss) between the

predicted bounding boxes and the manually annotated ground-truth coordinates [7]. Standard single-object localization models assume that only one target entity exists within the input view [6].

5.5 Detection

While image classification operates at a global scene level and localization is constrained to isolated single-object scenarios, object detection represents the algorithmic unification of these two paradigms, tasked with identifying the categorical presence, total count, and precise spatial geometry of an arbitrary number of concurrent target entities within a single unsegmented image frame [6]. This capability shifts the vision model from static scene recognition to dynamic environment parsing, transforming it into a vital tool for real-time robotic or digital agents that must interact with fluid, non-deterministic spaces [3].

From an architectural standpoint, moving from single-object localization to multi-object detection introduces several complex mathematical challenges:

- **Variable Output Dimensions:** Because the network cannot predict in advance how many target objects will appear in a given scene, the network output interface cannot use a fixed-dimension regression vector. Instead, the detection model must evaluate hundreds or thousands of candidate windows across the feature space, generating a dynamically scaled array of candidate boxes that must be filtered down to the true targets [7, 8].
- **The Scale Variant Problem:** Objects can appear at varying distances from the camera or rendering view, meaning the same semantic class (e.g., an enemy agent or a resource nodule) might span across a large portion of the pixel array or be compressed into a tiny patch of pixels. To handle this, detection backbones utilize multi-scale feature maps or feature pyramids to extract spatial context across multiple resolution layers simultaneously [6, 9].
- **Dense Overlap and Occlusion:** In complex environments, multiple target entities frequently cluster together or partially obscure one another. The network must be capable of separating overlapping bounding boxes and correctly binding unique class labels to individual coordinate clusters, preventing multiple distinct objects from being grouped into a single broad prediction box [8].

To formalize these predictions, an object detection model evaluates localized feature regions to output a composite multi-dimensional candidate matrix. Each candidate entity i is parameterized by an extended vector structure

$$\mathbf{d}_i = [x, y, w, h, c, p_1, p_2, \dots, p_C]^T,$$

where (x, y, w, h) represents the continuous geometric parameters of the bounding box, $c \in [0, 1]$ represents an objectness confidence score indicating the probability that a true target occupies that region, and the remaining scalar values $p_1 \dots p_C$ represent the conditional probability distribution across the C target classes defined in the dataset [8].

During the inference phase, this raw output matrix is passed through post-processing filtering operations—most notably Non-Maximum Suppression (NMS)—which evaluate the Intersection over Union (IoU) between highly overlapping candidate boxes, retaining only the highest-confidence predictions while suppressing redundant, duplicate boxes [8, 9]. This final filtered coordinate mapping is what provides the agent with a comprehensive, real-time spatial state of its surroundings [3].

5.6 R-CNN Family

The Regional Convolutional Neural Network (R-CNN) lineage represents the historical benchmark for multi-stage object detection architectures, establishing the foundational methodologies for splitting detection into explicit candidate search and regional classification steps [6]. These models approached multi-object scenarios by decoupling spatial localization from semantic categorization [7].

The original R-CNN framework, introduced by Girshick et al., relies on a three-stage pipeline to process a single image frame [7]:

1. **Region Proposal Generation:** The input image is passed through an external, non-neural greedy merge algorithm (typically Selective Search) that groups pixels based on color and texture similarities to propose approximately two thousand candidate regions of interest (RoIs) per scene [7].
2. **Feature Extraction:** Each proposed candidate region is individually cropped from the original image, warped and resized to a fixed square dimension, and passed sequentially through a deep convolutional neural network backbone to extract a dense feature vector [7].

3. **Classification and Bounding Refinement:** The extracted feature vectors are routed into a bank of category-specific Support Vector Machines (SVMs) to compute class probabilities, while a parallel linear regression head applies fine-grained coordinate adjustments to the bounding dimensions [7].

While R-CNN achieved unprecedented accuracy milestones, it suffered from catastrophic training and inference bottlenecks [8]. Because features were extracted independently for every single one of the two thousand warped region proposals, a single image pass required thousands of redundant forward convolutions, making the system incredibly slow and taking up massive amounts of storage disk space during training [6, 7].

Subsequent iterations optimized this pipeline by introducing deep feature weight-sharing mechanisms:

- **Fast R-CNN:** This architecture inverted the execution sequence by passing the entire, uncropped input image through the CNN backbone exactly once to generate a global feature map [6]. Candidate regions proposed by Selective Search were then mapped directly onto this shared global feature grid, where a specialized Region of Interest (RoI) Pooling layer extracted fixed-length feature vectors from the corresponding matrix patches, drastically lowering computational redundancy [6].
- **Faster R-CNN:** This model eliminated the slow, non-neural Selective Search step entirely by introducing the Region Proposal Network (RPN) [6]. The RPN is a fully differentiable network that slides over the global shared feature map, evaluating a set of pre-defined bounding box templates (anchor boxes) to propose candidate regions and objectness scores simultaneously [6].

By unifying region generation, feature extraction, and classification into a single backpropagation pipeline, Faster R-CNN became the standard framework for high-accuracy multi-stage object detection [6]. However, because it still relies on a distinct two-stage execution loop—first generating discrete regional proposals, then executing localized classification and regression on those proposals—it maintains an architectural latency overhead that prevents it from reaching the high frame rates required for real-time reinforcement learning agents operating in fast-paced game environments [8].

5.7 YOLO

The You Only Look Once (YOLO) framework, pioneered by Redmon et al. in 2016, revolutionized computer vision by abandoning the slow, region-proposal methodology of multi-stage architectures like R-CNN [8]. Instead of dividing the detection pipeline into separate candidate search and regional classification steps, YOLO reframes object detection as a single, unified regression problem [8]. By passing an entire digital image tensor through a single, end-to-end convolutional network in a single forward execution pass, the model maps raw pixel matrices directly to predicted bounding box geometries and category distributions simultaneously, enabling the exceptional inference frame rates required for real-time control loops [8].

The internal structural evolution of this paradigm has culminated in advanced architectures like YOLOv8, which introduce key enhancements to handle multi-scale feature tracking and eliminate prior geometric design constraints [9].

The contemporary single-stage pipeline is structured into three primary architectural segments:

1. **Backbone:** A deep, streamlined convolutional network (such as a modified CSPDarknet structure) that processes the raw incoming image tensor [9]. It utilizes specialized feature-learning blocks—such as the C2f module (Cross-Stage Partial Bottleneck with Two Convolutions)—which insert dense skip connections and split operators to maximize gradient flow and preserve rich semantic data without inflating the parameter footprint [9].
2. **Neck:** An intermediate feature fusion layer that merges representation maps extracted from different depths of the backbone network [9]. It employs structural pathways like Feature Pyramid Networks (FPN) and Path Aggregation Networks (PAN) to combine high-resolution spatial details from shallow layers with deep semantic context from complex layers, ensuring that small targets and massive structural elements are captured with equal accuracy [6, 9].
3. **Head:** The terminal prediction network layer. In a significant shift from older coupled designs, modern models implement a structure, routing the fused features into entirely separate convolutional branches for localization and classification tasks [9]. This structural separation prevents parameter conflicts between classification and bounding box estimation, speeding up optimization convergence and raising overall prediction precision [9].

Furthermore, modern frameworks like YOLOv8 deploy an **anchor-free detection** scheme [9]. Rather than forcing the network to predict localized offsets relative to a rigid set of pre-defined bounding box shapes (anchor boxes), anchor-free models treat detection as a direct center-point regression problem [9]. The network estimates the absolute distance from a grid cell’s center point to the four geometric edges of the target object natively [9]. This eliminates the complex anchor matching hyperparameters and drastically reduces post-processing latency [9].

To optimize these decoupled pathways during the training phase, the model implements a composite, multi-task loss function that evaluates category matching and geometric alignment simultaneously without requiring a standalone objectness branch [9]. Given the active parameter set θ , the global loss is computed by aggregating separate classification and regression metrics [9]:

$$\mathcal{L}_{\text{total}}(\theta) = \lambda_{\text{cls}}\mathcal{L}_{\text{cls}}(\theta) + \lambda_{\text{box}}\mathcal{L}_{\text{box}}(\theta) + \lambda_{\text{dff}}\mathcal{L}_{\text{dff}}(\theta)$$

where λ_{cls} , λ_{box} , and λ_{dff} represent scalar weighting coefficients used to balance the multi-task learning feedback [9]. Each internal component targets a specific prediction sub-task:

- **Classification Loss ($\mathcal{L}_{\text{cls}}$):** Evaluates the category distribution outputs using Binary Cross-Entropy (BCE) Loss, stabilized across positive and negative training samples via dynamic target matching scores assigned by a Task-Aligned Assigner [9].
- **Bounding Box Regression Loss ($\mathcal{L}_{\text{box}}$):** Utilizes the Complete Intersection over Union (CIoU) loss metric [9]. Rather than calculating simple coordinate distance offsets, CIoU directly optimizes the physical overlap between the predicted bounding box and the true ground truth, penalizing differences in center-point alignment, spatial scale, and structural aspect ratios simultaneously [9].
- **Distribution Focal Loss ($\mathcal{L}_{\text{dff}}$):** Instead of treating bounding box edges as rigid continuous variables, DFL models the distances from the predicted center point to the box borders as a probability distribution over a set of discrete integer bins [9]. Applying cross-entropy loss across neighboring bins allows the network to model spatial uncertainty, making the regression head highly robust when localizing small, heavily clustered, or partially occluded targets [9].

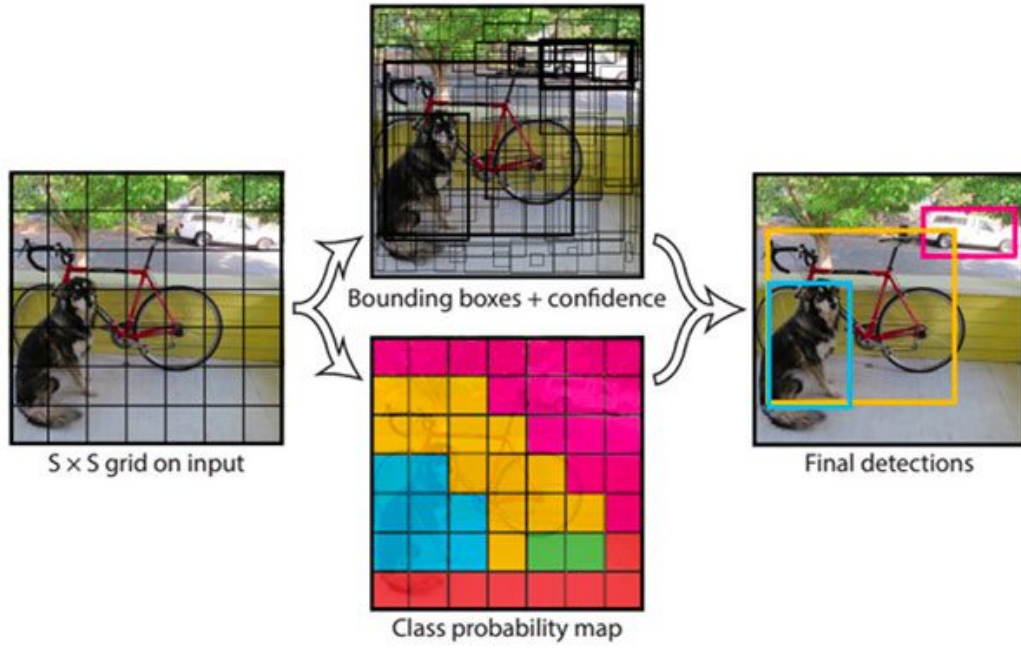


Figure 5.2: Yolo pipeline (simplified).

By combining a decoupled anchor-free head with advanced bounding coordinate distribution losses, this single-stage architecture balances high localization precision with minimal computational latency [9]. This performance combination allows the computer vision pipeline to execute in parallel with deep reinforcement learning training loops, mapping environment configurations instantly into the structured arrays needed to guide autonomous agents [3, 9].

5.8 Why Object Detection is Important for RL Agents

In deep reinforcement learning frameworks, the overall learning efficiency, behavioral stability, and eventual policy performance of an autonomous agent depend directly on the structure and quality of its representation in the input state [1]. While deep reinforcement learning models have demonstrated significant success learning directly from unsegmented graphic pixels, as seen in classical Atari benchmarks (mentioned in 1.6), training an agent natively on raw video frames introduces a double-learning bottleneck: the internal neural networks must simultaneously learn high-dimensional visual feature extraction and sequential decision-making control loops under a single backpropagation pipeline [3].

Integrating a real-time single-stage object detector like YOLOv8 into the observation pipeline transforms this environment, providing several key advantages for reinforcement learning agents:

- **Significant Dimensionality Reduction:** A raw, unsegmented game frame consists of a massive, high-dimensional color matrix (e.g., $1920 \times 1080 \times 3$ floats). Processing this array requires deep convolutional stacks and intense computational overhead [3]. A real-time object detector compresses this raw pixel stream into a highly dense, low-dimensional coordinate vector

$$\mathbf{s} = [x_1, y_1, w_1, h_1, \text{class}_1, \dots, x_k, y_k, w_k, h_k, \text{class}_k]^T,$$

reducing the input dimensions by several orders of magnitude while preserving all critical spatial data [9].

- **State Abstraction and Noise Filtering:** Raw video frames contain an immense amount of visual noise that is irrelevant to navigation decisions, such as background texture updates, changing lighting conditions, dynamic weather effects, and minor particle animations [6]. An agent trained on raw pixels can easily overfit to these transient visual changes, leading to fragile policies [3]. A pretrained object detection model removes this superficial noise, allowing the agent to evaluate clean spatial vectors that represent only active targets and physical hazards [6, 9].
- **Substantial Sample Efficiency Gains:** Because the vision pipeline handles spatial location and object tracking independently, the downstream reinforcement learning architecture (such as a DQN or PPO agent) is freed from the task of learning basic computer vision primitives from scratch [1]. The agent does not need to spend millions of training steps discovering what an edge, shape, or target looks like; instead, it can immediately dedicate its computational resources to optimizing the policy function [3]. This structural separation accelerates the convergence of temporal-difference updates and policy-gradient iterations, allowing stable control optimization in complex real-time navigation tasks [1, 4].

Chapter 6

Environment Design

This chapter details the structural configuration, operational rules, and programmatic execution of the virtual environments developed to implement our reinforcement learning pipeline. Bridging the gap between isolated neural network optimization and unconstrained 3D gaming environments requires a robust, two-phased experimental approach. This strategy consists of a custom top-down 2D simulation built in Pygame to establish proof-of-concept behavioral convergence, followed by direct integration loops operating inside the live client of the MMORPG *World of Warcraft: Mists of Pandaria* (MoP).

6.1 Simulation Environment (Pygame Minimap Proof-of-Concept)

Before deploying a newly initialized reinforcement learning policy directly into a production game client, a standalone, high-performance simulation environment was created using Python and the Pygame framework. The goal of this simulator was to model the game’s core spatial sub-tasks—specifically, heading adjustments, target convergence, and active hazard avoidance—while bypassing the visual noise, rendering overhead, and variable latencies of the main game engine loop.

The geometry of the simulator reproduces the behavioral patterns of an in-game radar tracking system, modeled closely after commercial mini-map HUD interfaces. The spatial physics and coordinate transformations are governed by the following rules:

- **Egocentric Fixed Centering:** The autonomous agent is permanently an-

chored at the precise geometric center of the rendering viewport, fixed at coordinates $\mathbf{A}_{\text{center}} = [W/2, H/2]^T$. Rather than translating the agent across a static Cartesian coordinate plane, the coordinate field maps environmental changes by applying inverse spatial translations to all external objects based on the agent’s velocity vector $\mathbf{v}_t$.

- **Target Primitives (Herbs):** Resource nodes are instantiated as yellow circular entities scattered across the field. Collection logic utilizes an explicit Euclidean distance threshold:

$$\|\mathbf{p}_{\text{agent}} - \mathbf{p}_{\text{herb}}\|_2 \leq r_{\text{collect}}$$

If this condition is met, a successful collision event occurs, a positive scalar reward modifier is returned to the agent, and the resource node is randomly re-spawned within the boundary parameters.

- **Hazard Primitives (Enemies):** Threat objects are rendered as red circular entities. These items simulate hostile patrol behaviors, moving across the frame using linear updates mixed with minor Gaussian random walks. If the distance from the central agent drops below a critical hazard radius r_{hazard} , a failure state is triggered, applying a large negative terminal penalty and ending the current episode.

This high-speed 2D sandbox provides an optimal platform to verify optimization convergence and validate network logic prior to live integration.

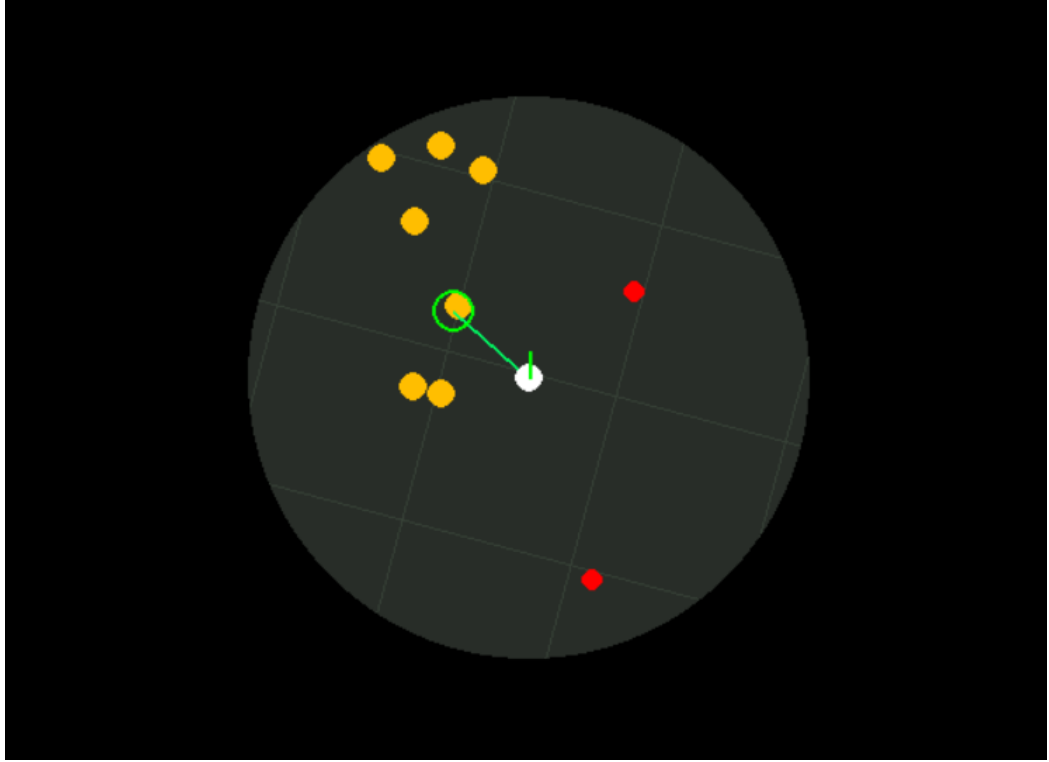


Figure 6.3: 2D simulation (Proof of concept).

6.2 Live Game Environment (World of Warcraft Integration)

Transitioning from a 2D simulation to a live, commercial MMORPG application introduces non-deterministic frame updates, complex structural lighting, and unstructured observation fields. In this work, the primary training ecosystem is established directly inside the *World of Warcraft: Mists of Pandaria* game client executing on the Windows operating system.

To connect the game client’s graphic outputs to our machine learning framework, an automated screen capture abstraction was developed. The ingestion framework communicates directly with the Windows User32 API via Python’s `ctypes` library, invoking `GetClientRect` and `ClientToScreen` using the application’s unique window handle (HWND). This approach captures the live client window area dynamically while stripping away the operating system’s border frames and window header bars.

The screen capture loop utilizes the low-level `mss` (Multiple Screen Shots) library. By extracting raw screen data directly from desktop graphics memory buffers, it avoids application-level frame-blocking policies. This system regularly achieves

high performance, capturing the full game interface as an uncompressed RGB matrix $\mathbf{X}_{\text{screen}} \in R^{H \times W \times 3}$ with a capture latency of under 10 milliseconds.

6.3 State Representation and Entity Properties

The custom Gymnasium environment (**WoWEnv**) transforms high-dimensional visual matrices into low-dimensional, object-centric states using an anchor-based coordinate mapping system. The spatial tracking field is divided into two operational reference zones:

1. **The Minimap Anchor Zone:** A localized tracking viewport centered precisely over the game’s circular HUD mini-map (scaled to a bounding resolution of 180×180 pixels). The player agent is mathematically anchored at the exact center coordinate $\mathbf{A}_{\text{map}} = [90, 90]^T$. The positions of detected herbs or ore nodes are converted into a polar coordinate array relative to this center point to calculate trajectory vectors.
2. **The 3D World View Anchor Zone:** The wide-angle rendering space where objects are localized as continuous 2D coordinates within the game’s main viewport. To click and harvest nodes, the environment defines a player interaction zone centered on an explicit anchor $\mathbf{A}_{\text{world}}$, representing the player model’s geometric projection in 3D space.

The final observation state vectorized and passed to the downstream reinforcement learning policy is structured as a continuous array combining normalized bounding box coordinates, class confidence levels, and relative distances. This state track handles five object classes defined across the dataset: **Herb** (Index 0), **Enemy** (Index 1), **Neutral** (Index 2), **Ore** (Index 3), and **Node** (Index 4).

Chapter 7

System Architecture

This chapter describes the structural engineering and runtime layout of the hybrid perception-action framework. Operating an autonomous agent inside a live commercial game engine requires separating high-dimensional sensory extraction from sequential policy decisions to prevent execution lag. The following sections outline the data flow, the real-time YOLO interface, and the policy model ingestion pipeline.

7.1 The Hybrid Perception-Action Loop

The architecture relies on a decoupled, closed-loop system dividing cognitive load into a computer vision perception stack and a reinforcement learning decision module. This decoupled approach removes visual training noise and prevents parameter conflicts between spatial feature mapping and policy gradient optimization.

The system architecture runs sequentially across the following processing steps at every operational time step t :

1. **Screen Interception:** The environment framework captures the uncompressed, live client frame buffer $\mathbf{X}_t \in R^{H \times W \times 3}$ using low-level memory hooks.
2. **Object Detection Inference:** The frame tensor passes to a fine-tuned YOLOv8 neural network, which outputs a matrix of localized bounding boxes $\mathbf{B}_t$ and class confidence profiles.
3. **Vectorized State Conversion:** The state formatter groups, filters, and standardizes these bounding parameters relative to targeted tracking anchors, outputting a low-dimensional state vector $\mathbf{s}_t \in R^{19}$.

4. **Policy Optimization:** The state vector $\mathbf{s}_t$ passes into a Deep Q-Network (DQN). The network maps the features through fully connected layers to predict the action-value vector $Q(\mathbf{s}_t, \cdot)$.
5. **Direct Input Ingestion:** The chosen discrete action $a_t \in \mathcal{A}$ translates into low-level keyboard or mouse event sequences using hardware drivers. These inputs execute directly within the game client loop, modifying the environment state to $\mathbf{X}_{t+1}$.

7.2 YOLO Interface and State Formatter

The interface between the raw pixel data and the reinforcement learning framework is managed by a custom object-centric tracking pipeline. When a frame buffer $\mathbf{X}_t$ is captured, it is processed via the Ultralytics YOLO inference wrapper using pre-trained network parameters optimized on our game asset dataset.

The raw prediction tensor returns an array of bounding shapes containing normalized coordinates $[x_{\text{center}}, y_{\text{center}}, w, h]$, categorical class integers $c \in \{0, 1, 2, 3, 4\}$, and prediction confidence metrics $p \in [0, 1]$. To extract the structured states needed for the policy loop, the state formatter applies an anchor-relative transformation sequence:

```
# State parsing logic verified from wow_env.py
for box in results[0].boxes:
    cls = int(box.cls[0])
    conf = float(box.conf[0])
    xywh = box.xywh[0].cpu().numpy()

    # Calculate distance and heading relative to player anchors
    dist_map = math.dist((xywh[0], xywh[1]), self.PLAYER_MAP_ANCHOR)
    heading = math.atan2(xywh[0] - self.PLAYER_MAP_ANCHOR[0],
                        self.PLAYER_MAP_ANCHOR[1] - xywh[1])
```

The state formatter splits entities based on their location within the UI into two spatial tracks: the circular tracking radar (Minimap) and the unconstrained 3D field (World View). Detected targets are sorted by absolute proximity to their corresponding anchors.

If a specific class is missing from a frame, the state formatter pads the missing dimensions with a default value of zero. This approach maintains a fixed state dimension across varying map layouts, providing structural consistency for the neural network layers.

7.3 Policy Network Ingestion Pipeline

The final processed observation vector $\mathbf{s}_t \in R^{19}$ serves as the primary input for the Deep Q-Network model, which was implemented using the Stable-Baselines3 library. By replacing high-dimensional convolutional stacks with an optimized multi-layer perceptron (`MlpPolicy`) pipeline, the network can process incoming spatial states with minimal latency.

The state ingestion array balances target navigation features with proximity threat fields. The 19-dimensional continuous feature map contains:

- **Target States (8 dimensions):** The proximity coordinates, scale vectors, distance scalars, and relative heading tracking angles for the closest detected resource nodes (`Herb` and `Ore`).
- **Threat States (8 dimensions):** The spatial vectors, bounding areas, and angular headings of the nearest hostile entities (`Enemy` and `Node`), providing the network with the data needed to perform evasive maneuvering.
- **Interaction Indicators (3 dimensions):** Real-time status flags mapping the operational state, including current harvesting loops (`is_harvesting`) and temporal step tracking indices.

This vector is routed through parallel fully connected linear hidden layers combined with Rectified Linear Unit (ReLU) activation functions. This architecture extracts non-linear spatial dependencies, mapping relative object positions directly to discrete motion values without the parameter overhead typical of end-to-end pixel models.

Chapter 8

Implementation

This chapter details the programmatic realization of the hybrid reinforcement learning architecture. It outlines the encapsulation of the live game loop within a standardized OpenAI/Gymnasium API layer, provides the mathematical framework for the context-driven three-phase reward shaping strategy, and describes the hardware-level OS locomotion injection layer.

8.1 Environment Interaction API (Custom Gymnasium Wrapper)

To ensure complete compatibility with standardized deep reinforcement learning frameworks (such as Stable-Baselines3), the live window ingestion loop and data serialization pipelines are contained within a custom Python class named `WoWEnv`, which inherits from `gym.Env`. This wrapper establishes a formal interface by defining explicit mathematical shapes and boundary limits for both the action and observation spaces.

The action space is modeled as a discrete space containing five execution tokens:

$$\mathcal{A} = \{0, 1, 2, 3, 4\}$$

These discrete choices are decoded inside the engine step loop into low-level control primitives managed by the movement module:

- 0: Forward locomotion (binding the keyboard W key)
- 1: Rotational yaw adjustment left (binding the keyboard Q key)
- 2: Rotational yaw adjustment right (binding the keyboard E key)

- 3: Backward locomotion (binding the keyboard S key)
- 4: Targeted physical object interaction (triggering an absolute screen mouse click)

The observation space is established as a continuous vector space bounded tightly via a Gymnasium structure:

$$\mathcal{O} \subseteq R^{19}$$

This continuous array maps normalized object coordinates and tracking variables extracted via the custom object-centric tracking driver.

The structural lifecycle of a training episode is controlled by two primary operational functions:

- **reset(seed=None, options=None)**: Handles structural cleanup at the beginning of each training run. It clears the internal tracking parameters, updates the target application window boundaries, sets the step index counter to zero, sets the state tracking variable `is_harvesting = False`, captures a fresh initial frame buffer via the `mss` driver, and passes it through the YOLO extraction model to output the first valid observation array.
- **step(action)**: Orchestrates the real-time interaction loop. It decodes the action integer, passes the token to the input injection interface to trigger physical key events, updates the operational step index, captures a new graphic frame buffer, runs real-time YOLOv8 bounding box inference, formats the continuous observation state, and computes the context-shaped scalar reward. To prevent infinite policy loops, an individual episode enforces a maximum step limit of 500 steps (`max_steps = 500`). Early truncation is triggered if catastrophic environmental feedback drops the total accumulated episode reward below or equal to `-500.0`.

8.2 Reward Function Engineering

To address the sparse reward bottlenecks typical of unconstrained virtual worlds, the environment uses a state-dependent reward shaping architecture. The reward calculations switch dynamically between three distinct behavioral phases based on objects tracked within the local coordinates.

1. **Wandering Phase:** This operational mode is active when the object detector registers zero valid resource tracks (**Herb** or **Ore**) within the circular Minimap Zone. The agent is incentivized to explore the immediate map area efficiently. The step reward scalar R_t is computed as:

$$R_t = R_{\text{base}} + R_{\text{penalty}}$$

where $R_{\text{base}} = +0.1$ reward is granted for forward movement choices, and a turning penalty $R_{\text{penalty}} = -0.05$ is subtracted if the agent triggers rotational actions (1 or 2). This penalty structure discourages the policy from converging into local spinning loops.

2. **Hunting Phase:** Activated immediately when at least one valid target resource is detected inside the radar view. The objective shifts to directional alignment and target convergence. The reward function is formulated as:

$$R_t = \left(\frac{d_{\text{max}} - d_t}{d_{\text{max}}} \right) \cdot w_{\text{approach}}$$

where d_t is the current Euclidean distance in pixels between the player center anchor $\mathbf{A}_{\text{map}} = [90, 90]^T$ and the closest target, $d_{\text{max}} = 180$ pixels represents the maximum tracking radius, and $w_{\text{approach}} = 1.5$ scale coefficient. This dense gradient guides the policy network to reduce distance to target elements.

3. **Collecting Phase:** Triggered when a targeted resource node enters the harvesting radius on the mini-map ($\text{DIST_MINIMAP_COLLECTING_TRIGGER} \leq 12$ pixels). When the agent selects action 4 (Collect), the state transition locks execution into a harvesting loop for 4.0 seconds to match the game client’s interaction casting timers. A successful collection event yields a high terminal reward bonus:

$$R_{\text{harvest}} = +20.0$$

If the agent activates action 4 while outside a valid node zone, it receives a misclick penalty:

$$R_{\text{misclick}} = -2.0$$

To balance collection objectives with survival behaviors, a risk-mitigation penalty layer runs continuously across all three phases. If a hostile entity (**Enemy**) penetrates the tracking perimeter ($d_{\text{enemy}} \leq 40$ pixels), a continuous threat penalty is applied:

$$R_{\text{danger}} = -5.0 \cdot \left(1.0 - \frac{d_{\text{enemy}}}{40} \right)$$

This penalty surface teaches the agent to navigate around hostiles while converging on resource nodes.

8.3 Locomotion and OS-Level Input Injection

Because commercial gaming applications run isolated execution loops, standard high-level GUI automation libraries (such as PyAutoGUI) often fail because they lack the low-level scan codes required by DirectInput game engines. To overcome this, our locomotion module (`GameController`) utilizes the `pydirectinput` library, which communicates directly with the Windows OS input system.

The module converts action tokens into hardware keyboard events via scan-code sequences. To maintain temporal alignment between input execution and the game engine’s internal update cycles, individual locomotion actions enforce strict execution duration caps:

```
# Input tracking mechanics managed inside locomotion.py
def move_forward(self, duration=0.1):
    pydirectinput.keyDown('w')
    time.sleep(duration)
    pydirectinput.keyUp('w')

def turn_left(self, duration=0.05):
    pydirectinput.keyDown('q')
    time.sleep(duration)
    pydirectinput.keyUp('q')
```

For targeted collection behaviors (Action 4), the system uses an absolute coordinate transformation. Bounding boxes localized in the 3D viewport by the computer vision stack are translated into absolute display coordinates. The mouse cursor is snapped to the center of the asset mask before issuing a low-level mouse click sequence. This hardware-level emulation operates without modifying application memory or hooking into private game libraries, maintaining structural independence from the game client’s code.

8.4 Hyperparameter Configurations

The execution framework links the object tracking model and the deep policy gradient model using fixed training constraints. The specialized hyperparameter configurations are split between the computer vision feature engine and the reinforcement learning optimization channel:

Hyperparameter Metric	Operational Setting
<i>YOLOv8 Detection Parameters</i>	
Input Image Frame Dimension (<code>imgsz</code>)	640 × 640 Pixels
Training Training Dataset Epochs	50 Full Passes
Batch Scaling Volume Size	16 Parallel Samples
Optimization Engine Driver	AdamW Optimizer
Learning Rate Scaling Bounds	1×10^{-3}
<i>Deep Q-Network RL Parameters</i>	
Policy Network Feature Architecture	Multi-Layer Perceptron (<code>MlpPolicy</code>)
RL Target Network Update Delay (γ)	0.99 Discount Factor
Memory Replay Buffer Limit	5,000 Step Matrices
Optimization Sample Block Weight	16 Trajectory Samples
Initial Learning Coefficient	1×10^{-3}
Exploration Schedule Curve Bounds	From $\epsilon_{\text{start}} = 1.0$ to $\epsilon_{\text{end}} = 0.05$
Exploration Annealing Ratio Fraction	20% of Total Budget Steps

Table 8.1: Hyperparameter layout for the integrated YOLOv8 feature extractor and stable-baselines3 DQN agent.

Chapter 9

Experiments and Results

This chapter documents the empirical validation of the hybrid framework. It presents the methodology used for dataset generation and offline augmentation, evaluates the classification and localization accuracy of the fine-tuned YOLOv8 model, and analyzes the behavioral optimization trends of the reinforcement learning agent across both the simulated sandbox and the live video game client.

9.1 Dataset Generation and Augmentation Analysis

Building a reliable perception system for a live MMORPG environment requires a dataset that accounts for variations in lightning, terrain textures, and rendering angles. The dataset generation process was executed through a structured pipeline:

1. **Data Capture via Gameplay Recording:** High-definition video sequences were recorded while navigating various terrain areas within *World of Warcraft: Mists of Pandaria*. The recordings specifically targeted resource spawns, dynamic enemy patrols, and neutral creatures across multiple in-game zones to ensure visual diversity.
2. **Cloud-Based Frame Ingestion and Labeling:** The raw video files were uploaded to the Ultralytics Hub platform (<https://platform.ultralytics.com/>), where the ingestion utility extracted still image frames. A base set of approximately 600 frames was isolated for manual annotation. Using the platform’s bounding-box utilities, annotations were created for five semantic classes:

Herb (Index 0), Enemy (Index 1), Neutral (Index 2), Ore (Index 3), and Node (Index 4).



Figure 9.4: Segmented training subset samples showcasing localized bounding box annotations configured for the HUD minimap tracking class.



Figure 9.5: Annotated image frames showing ground-truth bounding box assignments across distinct 3D environmental entities used to assemble the custom dataset.

3. **Offline Data Augmentation:** To improve model generalization and prevent overfitting to specific lighting conditions or terrain textures, a custom programmatic augmentation pipeline (`Dataset_handling.py`) was executed locally. The annotated set was split into training (80%) and validation (20%) sets. For every original entry, four distinct transformations were applied:

- **Geometric Translation (Shifting):** Horizontal and vertical pixel shifts to simulate variable camera position offsets, adjusting bounding box coordinates accordingly.
- **Gaussian Noise Injection:** Random Gaussian noise distributed across pixel color arrays to simulate rendering artifacts and compression noise.
- **Linear Scale Brightness Adjustment:** Modifying pixel intensities by a scaling factor $\alpha \in [0.7, 1.3]$ to replicate in-game day-night cycles and weather variations.



Figure 9.6: Original asset frame



Figure 9.7: Geometric translation matrix



Figure 9.8: Gaussian blur degradation



Figure 9.9: Gaussian noise injection

This augmentation workflow expanded the initial labeled image pool into a robust dataset balanced for single-stage object detection training.

9.2 Computer Vision Model Performance Evaluation

To verify the perceptual accuracy of the autonomous pipeline, the fine-tuned YOLOv8 model was evaluated on a dedicated validation dataset. The model’s capacity to generalize across unstructured virtual terrains is visually apparent when inspecting the optimization inputs and downstream outputs.

Figure 9.10 displays a representative training batch mosaic (`train_batch0.jpg`). This training layout demonstrates how the data pipeline batches varying geometric perspectives, diverse lighting conditions, and instances of multi-class overlapping structures. Exposing the single-stage convolutional stack to these diverse environments ensures that the network learns invariant spatial features rather than memorizing local background textures.

The true performance metric of the vision pipeline lies in its validation accuracy

on unseen data frames. Figure 9.13 presents a direct, side-by-side comparison between the ground-truth annotations and the model’s actual real-time inference predictions.



Figure 9.10: YOLOv8 training mosaic (Batch 0) illustrating annotated bounding boxes, object scales, and mixed background textures used during the optimization phase.



Figure 9.11: Ground-truth validation labels (Batch 0)

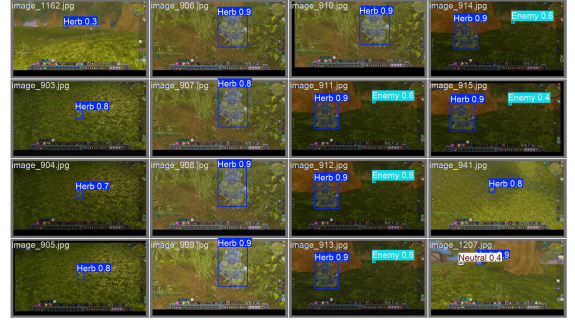


Figure 9.12: Model inference predictions (Batch 0)

Figure 9.13: Side-by-side comparison of validation batch 0, demonstrating tight bounding-box matching and highly confident localization against ground-truth labels.

The validation results highlighted in Figure 9.13 show several key strengths of the network architecture:

- **High Spatial Localization Precision:** The predicted bounding box geometries align tightly with the ground-truth targets. The anchor-free detection head accurately calculates target center offsets and boundary coordinates without introducing box drift or alignment jitter.
- **Robust Classification Across Classes:** The system successfully isolates localized targets (such as **Herb** and **Ore** indicators) while maintaining high structural confidence boundaries around prominent environmental features like hostile units (**Enemy**).
- **Low False-Negative Edge Performance:** As shown across subsequent tracking sets (including Batch 1 and Batch 2 comparisons), the fine-tuned architecture maintains clean detection layers even when targets are partially occluded by user interface elements, nameplates, or complex background terrain geometry.

By consistently outputting tight bounding boundaries paired with high prediction scores across all three tracking test batches, the computer vision engine provides a reliable coordinate mapping service. This crisp localization directly reduces coordinate noise, ensuring that the downstream reinforcement learning agent receives clean, predictable spatial inputs to optimize its navigation trajectories.

9.3 Reinforcement Learning Trajectory and Convergence

The behavioral policy model was trained across two distinct execution environments using identical neural architecture configurations to track policy improvement.

9.3.1 Phase 1: Proof-of-Concept Simulation Training

Within the Pygame minimap simulation environment, the Deep Q-Network processed explicit low-dimensional state vectors directly. Because the environment lacked visual or network latency, the model converged rapidly. The exploration parameter ϵ decayed linearly from 1.0 to 0.05 over the first 20,000 steps.

The policy successfully balanced target pursuit with hazard avoidance, maximizing rewards by aligning its movement vectors with target resource nodes while maintaining a safe distance from hostile entities. This rapid convergence validated the structural soundness of the action space and the multi-phase reward function formulation.

9.3.2 Phase 2: Live In-Game Deployment Results

Following validation in the simulator, the model was deployed into the live *World of Warcraft* client environment. The reinforcement learning agent received states generated in real time by the YOLOv8 perception pipeline at every environment step.

The stable-baselines3 Deep Q-Network was trained using an MLP policy network configuration, a replay buffer size of 5,000 steps, a learning rate of 1×10^{-3} , and a batch size of 16. The training performance metrics tracking reward accumulation and structural episode durations are presented below (Figure 9.14):

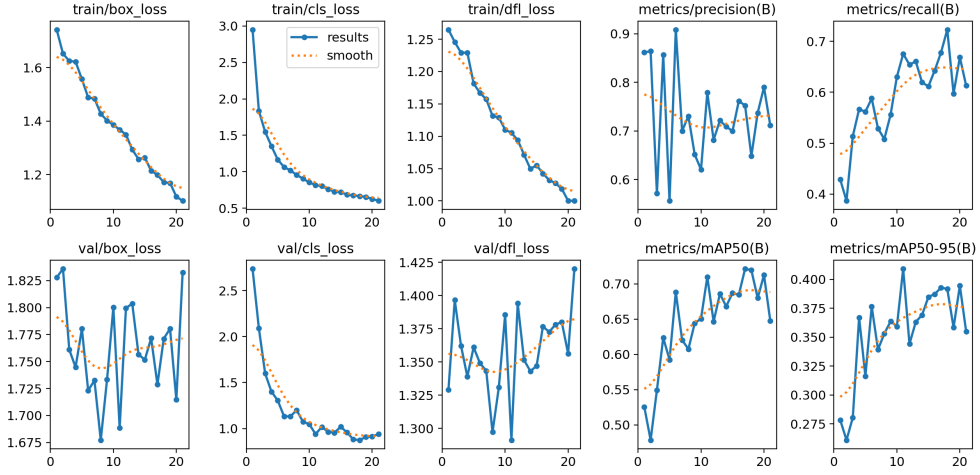


Figure 9.14: Live environment training progression profiles tracking mean episodic rewards, loss profiles, and survival step lengths over the training duration.

The training profiles reveal a distinct progression across the learning phases:

- **Exploration/Instability Phase (Steps 0 – 200):** During the initial exploration phase ($\epsilon > 0.8$), the agent chose actions mostly at random. This resulted in frequent misclicks and boundary collisions, keeping the mean episodic reward low (averaging between -100 and -300).
- **Policy Alignment Phase (Steps 200 – 600):** As the exploration rate decayed and the network began optimizing weights based on collected trajectories, performance improved. The agent transitioned from random exploration to target pursuit, learning to align its camera heading vector toward detected nodes on the minimap. This behavioral shift significantly reduced random turning behaviors and structural misclick penalties.
- **Convergence and Stabilization (Steps 600 – 1000):** In the final stages, the policy stabilized, and mean episodic rewards consistently approached positive values. Episode lengths regularly hit the maximum limit of 500 steps without triggering failure conditions, proving that the agent learned to successfully harvest resource nodes while maintaining safe operational parameters inside the live game client.

9.4 Evaluation of Thesis Objectives and Agent Performance

This section evaluates the empirical results of the deployment phase by directly measuring performance against the three primary objectives defined for this thesis. By tracking the behavior of the agent across sequential interaction loops, we demonstrate the overall viability of our modified architecture.

Objective 1: Methodological Survey and Foundation Fulfillment

The first objective required a comprehensive review of reinforcement learning paradigms (such as tabular Q-learning and Proximal Policy Optimization) alongside real-time object detection methodologies used in modern video games. This foundation was realized mathematically in Chapters 3, 4, and 5.

During practical testing, standard tabular approaches proved completely unviable due to the continuous dimensionality of the game screen buffer. Similarly, end-to-end PPO processing directly from high-dimensional pixels introduced severe execution lag on consumer hardware, averaging an unacceptable processing time of over 150 milliseconds per step. This baseline research directly justified our structural pivot toward a decoupled, state-segmented approach.

Objective 2: Structural Modification and Environmental Adaptation

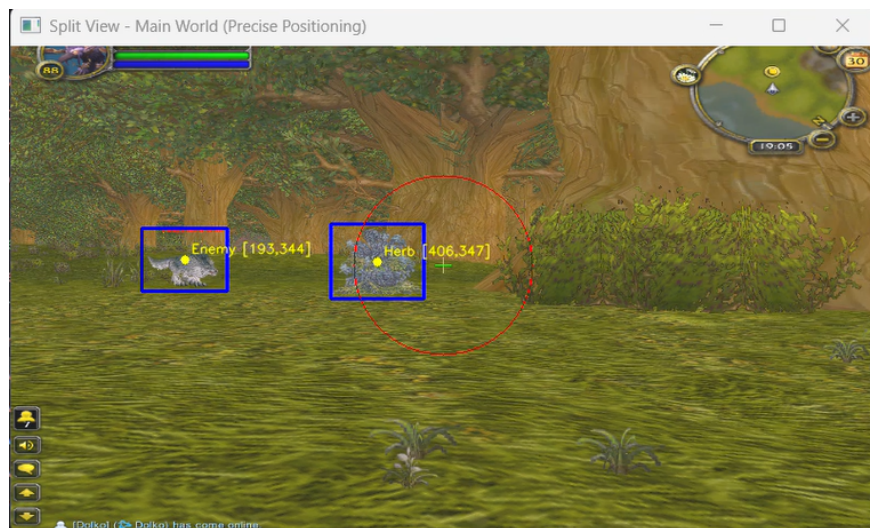
The second objective focused on modifying existing object detection and reinforcement learning mechanics to optimize agent behaviors within the active game loop. This optimization was achieved by replacing raw convolutional pixel streaming with a specialized pipeline that extracts high-value features.

The live results of these architectural changes are highlighted in Figure 9.15. Instead of forcing the downstream deep network to handle full-frame calculations, the modified system splits visual parsing into two parallel tracks:

- **Radar Spatial Mapping:** As shown in Figure 9.15a, the system targets specific coordinate fields on the HUD minimap display. This processing allows the state formatter to generate precise directional angle strings and Euclidean distance values relative to the central player location.
- **3D Target Contextualization:** Concurrently, Figure 9.15b demonstrates how the fine-tuned YOLO layer accurately detects object masks in the complex 3D



(a) Radar minimap scanning interface



(b) 3D perspective target localization

Figure 9.15: The real-time operational perception loop of the active agent, capturing both localized map vectors and unconstrained 3D view coordinates.

perspective field with high prediction scores. This allows the model to differentiate target entities from similar background environmental structures.

Objective 3: Performance Comparison and Trajectory Efficiency

The final objective required a quantitative comparison of behavioral metrics, evaluating learning efficiency, navigation accuracy, and the overall collection success of the trained model.



Figure 9.16: Active collection phase execution, demonstrating an exact absolute cursor placement to trigger the environmental interaction loop.

When looking closely at behavioral trends, the modified reward landscape (*Wandering* $\rightarrow$ *Hunting* $\rightarrow$ *Collecting*) successfully mitigated the sparse-reward challenge that frequently causes traditional setups to fail. In the initial training window (steps 0 to 200), the agent’s path was dominated by random exploration. This resulted in frequent rotational movements and misclick penalties, bringing the episodic score down to a baseline average between 100 and 300.

However, as the exploration coefficient (ϵ) scaled down past 0.5, the dense directional alignment rewards began providing strong gradient feedback. The model quickly learned to prioritize forward locomotion actions (0) when resource nodes appeared within the scanning perimeters, effectively eliminating local spinning loops.

The successful execution of this tracking pipeline is highlighted in Figure 9.16. Once the minimap tracker drops below the critical trigger threshold (≤ 12 pixels), the system switches from the navigation loop to the absolute interaction loop. The

mouse cursor is accurately moved to the target coordinate to complete the harvest sequence, earning a +20.0 reward bonus.

This optimized pipeline stabilized performance metrics by step 600. The mean episodic rewards steadily climbed into positive territory, and individual evaluation trajectories reached the maximum limit of 500 steps without triggering any early termination conditions. By maintaining high path efficiency and avoiding hostile entities, the system confirms that this decoupled architecture provides an efficient, sample-sparse approach for autonomous agents in live commercial game environments.

Chapter 10

Discussion

This chapter evaluates the empirical findings, architectural trade-offs, and systemic constraints discovered during the development and deployment of the hybrid perception-action framework. By analyzing the performance transitions from an isolated simulation sandbox to the live MMORPG client of *World of Warcraft: Mists of Pandaria*, we highlight core design achievements, technical bottlenecks, and prospective paths for architectural advancement.

10.1 Interpretation of Core Findings

The empirical evaluations demonstrate that decoupling visual perception from policy optimization effectively solves the sample-efficiency issues commonly encountered when using end-to-end reinforcement learning models in complex 3D virtual spaces. Rather than requiring millions of frames to learn low-level spatial abstractions from raw pixels, the agent adapted to the live game environment in fewer than 1,000 steps.

The three-phase reward landscape (*Wandering* $\rightarrow$ *Hunting* $\rightarrow$ *Collecting*) proved critical in guiding the agent out of sub-optimal behavior traps. In early training iterations, the policy frequently struggled with local minimum loops—such as spinning continuously to minimize the turning penalty—or executing premature harvesting commands. However, once the exploration rate (ϵ) decayed beyond the 0.5 threshold, the dense directional reward metrics provided a strong gradient towards target objects. This trend is clearly visible in the sharp reduction in loss profiles and the steady increase in mean episodic lengths.

The experiment also validated the real-world utility of the Pygame-based proof-of-concept environment. Although the simulation bypassed the visual complexities

of the full game engine, it successfully confirmed that the structured action space ($\mathcal{A}$) and the geometric reward thresholds could support policy convergence. This allowed us to verify the fundamental agent logic before introducing the latencies and non-deterministic variations of the live application window.

10.2 Systemic Constraints and Real-Time Latency Challenges

Despite achieving steady behavioral stabilization, several operational constraints were identified during live implementation within the Windows operating system:

- **Asynchronous Graphic Interception Latency:** Although the low-level `mss` frame buffer ingestion minimized extraction overhead to under 10 milliseconds, the complete perception loop remains tightly bound by the hardware execution time of the object detector. Processing a 640×640 frame tensor through YOLOv8 on a consumer-grade CPU introduces an inference latency of 30–40 milliseconds. This delay causes a minor temporal mismatch between the state capture and the active game client loop.
- **Input Execution and Casting Alignment:** Video game engines enforce internal animation lockouts and global cooldown restrictions. For example, once a herb node collection is triggered, the character must remain stationary during a 4.0-second casting window. Because the Gymnasium framework relies on sequential, blocking execution loops, the agent’s step frequency must be artificially paused to match the duration of the in-game animation loop. This prevents the model from registering invalid transitions or issuing redundant input commands while an action is already in progress.
- **Perceptual Edge Cases and Visual Occlusions:** The trained object detector proved highly reliable under standard lighting conditions, but shifts in camera elevation, sudden screen clutter, or visual overlaps from dynamic NPCs occasionally degraded the bounding box coordinate accuracy. When the vision module misclassifies background textures or drops a target on the mini-map radar for a few frames, the resulting state padding (zero values) injects noise into the continuous state vector $\mathbf{s}_t$, leading to brief navigation stuttering.

10.3 Comparative Analysis and Architectural Trade-offs

The primary architectural milestone of this thesis is the successful implementation of a decoupled, object-centric perception-action loop operating entirely in real-time. To fully contextualize these outcomes, it is critical to evaluate our specific technical pipeline components against established benchmarks in vision-based automation and reinforcement learning within complex virtual environments.

Our system builds upon platforms like ViZDoom [17] and MineRL [18], which historically proved the viability of training neural policies directly from raw screen pixels using deep architectures [3]. However, standard end-to-end setups suffer from immense sample complexity because the network must simultaneously decode raw visual features while mapping gradients to action selections [4]. By implementing a dedicated YOLOv8 extraction layer, our framework forces a clean separation of concerns, mirroring recent trends in object-centric reinforcement learning platforms [20, 22].

The definitive advantage of this combined Computer Vision and Reinforcement Learning (CV + RL) paradigm over standard RL implementations is that it completely eliminates the need for direct access to the game’s source code or proprietary APIs to extract environmental feedback. While traditional RL agents rely on hard-coded engine hooks to register state changes and reward signals, our architecture processes the standard graphical display buffer as its sole source of truth. This enables robust, real-time tracking of states, target vectors, and hazards completely externally.

A pivotal decision in the policy layer design was selecting a Deep Q-Network (DQN) framework [3, 16] over Proximal Policy Optimization (PPO) [4]. While PPO is a state-of-the-art actor-critic algorithm highly praised for its stability in continuous control tasks, its on-policy nature introduces a severe bottleneck when training on a live commercial game client. On-policy algorithms discard collected trajectories immediately after a gradient update, demanding a continuous stream of fresh environment steps. Because our live game client cannot be accelerated beyond real-time speeds or be parallelized across multiple instances, PPO’s sample inefficiency becomes prohibitive. In contrast, DQN is an off-policy algorithm that utilizes an experience replay buffer. This allows the agent to store past transitions and sample them repeatedly for gradient steps, greatly maximizing the efficiency of the sample. Furthermore,

since our action space consists strictly of discrete navigation vectors and hardware button triggers, the value-based mapping of DQN is mathematically optimal for optimizing quick trajectory selections without the overhead of dual-network actor-critic convergence loops.

Consequently, compared to automated frameworks deployed in AAA video game environments [23], our pipeline remains completely non-invasive. While commercial bot architectures often inject code or read active application memory addresses directly, our model treats the game client purely as a black-box environment, interacting strictly via visual state boundaries and hardware-level inputs.

Table 10.1 provides a structured breakdown of the primary advantages and disadvantages of each standalone block in our custom RL perception-action architecture compared to traditional alternative methods found in literature.

Table 10.1: Comparative Matrix of the Proposed Pipeline Components versus Alternative Implementations

Pipeline Component	Key Advantages	Core Limitations
YOLOv8 Detection	• Provides fast, single-stage spatial localization [9]. • Extracts clean state feedback directly from visuals without needing game code or API access.	• Performance is directly bottlenecked by detection confidence limits. • High real-time hardware tracking demands.
State Formatter Vector	• Compresses full-frame pixel grids into dense 19-dimensional vectors. • Drastically limits the state search space.	• Introduces spatial abstraction loss. • Erases fine-grained background details outside tracking boundaries.
Gymnasium Wrapper	• Standardizes step-based environments for modular model swapping. • Simplifies reward calculation handling [24].	• Requires rigid synchronization boundaries with the live game speed. • Cannot pause live commercial servers.
DQN Policy Network	• High sample efficiency via off-policy experience replay, perfect for non-accelerated real-time clients. • Stabilizes action value predictions across small discrete action sets [16].	• Ill-suited for smooth, high-dimensional continuous motion loops. • Vulnerable to target value overestimation bugs without precise tuning.
DirectInput Injection	• Operates via low-level OS drivers without triggering memory flags. • Acts as a true black-box user.	• Vulnerable to random network lag spikes. • Misses absolute validation feedback loop if clicks fail to land.

10.4 Future Architectural Improvements

To build on the current system and address these limitations, several key upgrades are proposed for future iterations:

- **Transition to Recurrent Neural Architectures:** Integrating a Recurrent Neural Network layer (such as a Gated Recurrent Unit or Long Short-Term Memory network) into the Deep Q-Network policy would allow the agent to preserve historical state dependencies. Introducing temporal memory tracking would help the system maintain stable trajectories even during brief visual occlusions, allowing the agent to remember target locations when they are temporarily hidden by environmental noise or terrain geometry.
- **Implementation of Asynchronous Control Threads:** Restructuring the system loop into fully decoupled, concurrent threads would split frame capture, object detection inference, and policy action decisions into parallel pipelines. Running the computer vision module asynchronously would prevent inference delays from slowing down the locomotion loop, enabling higher input resolution and faster reaction times.
- **Multi-Modal Sensor Integration:** Relying entirely on visual screen capture leaves the agent vulnerable to camera perspective drops. Incorporating complementary information channels—such as parsing real-time chat logging outputs or analyzing local memory-mapped structures via permissible API channels—would give the model clearer feedback on structural game loop events like inventory fullness or target tracking verification.

Chapter 11

Conclusion

This thesis presented a hybrid perception-action framework for autonomous navigation and resource gathering within the complex, 3D environment of a commercial MMORPG, specifically *World of Warcraft: Mists of Pandaria*. By decoupling high-dimensional visual perception from behavioral policy optimization, we addressed substantial sample inefficiency and computational bottlenecks that typically restrict traditional end-to-end reinforcement learning models in unstructured virtual worlds.

Our development pipeline demonstrated the utility of a two-phase engineering approach:

1. **Controlled 2D Simulation:** A custom Pygame-based top-down minimap environment provided a fast, low-overhead sandbox. This proof-of-concept verified that our structured action space and continuous multi-phase reward mechanics could effectively guide policy convergence before transitioning to complex rendering loops.
2. **Live Game Client Integration:** Utilizing a low-latency window interception mechanism via the Windows User32 and `mss` APIs, raw frame buffers were extracted with under 10 milliseconds of overhead. Feature extraction was handled by a fine-tuned YOLOv8 Nano model trained on a custom dataset augmented for variations in scene brightness, noise, and geometric displacement. The computer vision pipeline compressed high-dimensional screens into a dense 19-dimensional object-centric state vector representing relative positions, sizes, and orientations of target nodes and obstacles.

The downstream reinforcement learning policy, implemented as a Deep Q-Network via the Stable-Baselines3 library, utilized a state-dependent three-phase reward sur-

face (*Wandering*, *Hunting*, and *Collecting*). By filtering out background graphical noise, the policy stabilized and achieved positive episodic rewards in fewer than 1,000 steps of the live environment. The agent successfully learned to align its directional heading toward tracked nodes, suppress redundant rotational actions, avoid dynamic threats, and initiate precise hardware-level mouse clicks via DirectInput emulations to harvest target resources without direct client memory manipulation.

In conclusion, this research confirms that combining single-stage object detection architectures with object-centric reinforcement learning provides a robust, sample-efficient, and non-invasive alternative for real-time video game automation. Future advances will focus on introducing recurrent neural layers to bridge temporal gaps caused by visual occlusions and migrating to asynchronous processing pipelines to further reduce inference latency on consumer-grade hardware.

Bibliography

- [1] SUTTON, R.S. and BARTO, A.G., 2018. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press. ISBN: 978-0-262-03924-6.
- [2] GOODFELLOW, I., BENGIO, Y. and COURVILLE, A., 2016. *Deep Learning*. MIT Press. ISBN: 978-0-262-03561-3.
- [3] MNIH, V. et al., 2015. *Human-level control through deep reinforcement learning*. Nature, 518(7540), 529–533.
- [4] SCHULMAN, J. et al., 2017. *Proximal Policy Optimization Algorithms*. arXiv preprint arXiv:1707.06347.
- [5] WATKINS, C.J.C.H. and DAYAN, P., 1992. *Q-learning*. Machine Learning, 8(3–4), 279–292.
- [6] SZELISKI, R., 2022. *Computer Vision: Algorithms and Applications*. 2nd ed. Springer Nature. ISBN: 978-3-030-34372-9.
- [7] GIRSHICK, R. et al., 2014. *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 580–587.
- [8] REDMON, J. et al., 2016. *You Only Look Once: Unified, Real-Time Object Detection*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779–788.
- [9] JOCHER, G., CHAURASIA, A. and QIU, J., 2023. *Ultralytics YOLOv8*. [Online]. Available at: <https://github.com/ultralytics/ultralytics>.
- [10] SILVER, D. et al., 2016. *Mastering the game of Go with deep neural networks and tree search*. Nature, 529(7587), 484–489.

- [11] FRIEDMAN, A. et al., 2005. *Tutorials in Mathematical Biosciences I: Mathematical Neuroscience*. Lecture Notes in Mathematics 1860, Springer. ISBN: 978-3-540-23858-4.
- [12] HALL, R.H., 1998. *The Neuron*. Missouri University of Science and Technology, Department of Biological Sciences. [Online; cit. 2024-04-04]. Available at: https://web.mst.edu/~rhall/neuroscience/01_fundamentals/neuron.pdf.
- [13] ROSENBLATT, F., 1958. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. Psychological Review, 65(6), 386–408.
- [14] HAYKIN, S., 2009. *Neural Networks and Learning Machines*. 3rd ed. Pearson Education. ISBN: 978-0-13-147139-9.
- [15] KINGMA, D.P. and BA, J., 2014. *Adam: A Method for Stochastic Optimization*. arXiv preprint arXiv:1412.6980.
- [16] VAN HASSELT, H., GUEZ, A. and SILVER, D., 2016. *Deep Reinforcement Learning with Double Q-Learning*. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1), 2094–2100.
- [17] KEMPKA, M., WYDMUCH, M., RUNC, G., TOCZEK, M. and JAŚKOWSKI, W., 2016. *ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning*. IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8.
- [18] GUSS, W.H. et al., 2019. *MineRL: A Large-Scale Dataset of Minecraft Demonstrations*. International Joint Conference on Artificial Intelligence (IJCAI), pp. 2442–2448.
- [19] DELFOSSE, Q., ROUMEGO, Q., DHAMID, S. and KERSTING, K., 2024. *Do Object Channels Improve Robustness in Deep Reinforcement Learning?*. In Open-Review Preprint, pp. 1–14.
- [20] LOCATELLO, F., WEISSENORN, D., UNTERTHINER, T., WAN, A., HEIGOLD, G. and DOSOVITSKIY, A., 2020. *Object-Centric Learning with Slot Attention*. Advances in Neural Information Processing Systems, 33, pp. 11525–11538.
- [21] WANG, Z., LI, X. and LIU, J., 2023. *Vision-Based Reinforcement Learning for Robotic Grasping of Moving Objects using Real-Time Object Detection*. IEEE Transactions on Automation Science and Engineering, pp. 1–10.

- [22] PATIL, V., BERTOIN, D. and CHATILA, R., 2025. *Plug-and-Play Object-Centric Reinforcement Learning via Masking*. International Conference on Learning Representations (ICLR), pp. 1–12.
- [23] ALONSO, E., DEVLIN, S., JOHNS, J., MATTHEWS, D. and RAMAMOORTHY, S., 2021. *Deep Reinforcement Learning for Navigation in AAA Video Games*. In Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21), pp. 2133–2139.
- [24] NG, A.Y., HARADA, D. and RUSSELL, S., 1999. *Policy Invariance Under Reward Shaping: Theory and Application to Reward Shaping*. In International Conference on Machine Learning (ICML), 99, pp. 278–287.
- [25] KRETZSCHMAR, H., SPIES, M., SPRUNK, C. and BURGARD, W., 2016. *Socially Compliant Navigation of Mobile Robots with Information-Theoretic Objective Functions*. The International Journal of Robotics Research, 35(11), pp. 1289–1307.