UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Marcelo Luiz Harry Diniz Lemos

Towards a Scale-Invariant Reinforcement Learning Model for Real-Time Strategy Games

Belo Horizonte
2024

Marcelo Luiz Harry Diniz Lemos

Towards a Scale-Invariant Reinforcement Learning Model for Real-Time Strategy Games

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Luiz Chaimowicz
Co-Advisor: Anderson Rocha Tavares and
Leandro Soriano Marcolino

Belo Horizonte
2024

Lemos, Marcelo Luiz Harry Diniz.

L557t Towards a scale-invariant reinforcement learning model for real-time strategy games [recurso eletrônico] / Marcelo Luiz Harry Diniz Lemos - 2024.
1 recurso online (83 f. il., color.) : pdf.

Orientador: Luiz Chaimowicz.
Coorientadores: Anderson Rocha Tavares e Leandro Soriano Marcolino.
Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 77-83

1. Computação – Teses. 2. Aprendizado do computador – Teses. 3. Aprendizado profundo – Teses. 4. Jogos eletrônicos – Teses. 5. Jogos de estratégia (Matemática) – Teses.
I. Chaimowicz, Luiz. II. Tavares, Anderson Rocha. III. Marcolino, Leandro Soriano. IV. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação.
V. Título.

CDU 519.6*82(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS

TOWARDS A SCALE-INVARIANT REINFORCEMENT LEARNING MODEL FOR REAL-TIME STRATEGY GAMES

MARCELO LUIZ HARRY DINIZ LEMOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Luiz Chaimowicz - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Anderson Rocha Tavares - Coorientador
Instituto de Informática - UFRGS

Prof. Leandro Soriano Marcolino - Coorientador
LIRASC - Lancaster University

Prof. Erickson Rangel do Nascimento
Departamento de Ciência da Computação - UFMG

Prof. Levi Henrique Santana de Lelis
Department of Computing Science - University of Alberta

Belo Horizonte, 29 de agosto de 2024.

Documento assinado eletronicamente por **Luiz Chaimowicz**, **Professor do Magistério Superior**, em 04/11/2024, às 14:52, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Anderson Rocha Tavares**, **Usuário Externo**, em 09/12/2024, às 15:30, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Leandro Soriano Marcolino**, **Usuário Externo**, em 10/12/2024, às 12:51, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Levi Henrique Santana de Lelis**, **Usuário Externo**, em 13/12/2024, às 03:11, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Erickson Rangel do Nascimento**, **Professor do Magistério Superior**, em 13/12/2024, às 08:30, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3498991** e o código CRC **A36871E5**.

Referência: Processo nº 23072.248331/2024-66 SEI nº 3498991

Acknowledgments

I would like to express my sincere gratitude to my advisors, Luiz Chaimowicz, Leandro Soriano Marcolino, and Anderson Rocha Tavares, for their guidance, encouragement, and support throughout the duration of this project. Their expertise and commitment to my success have been instrumental in shaping the direction of this thesis.

I am also deeply grateful to my family and friends for their unwavering support and encouragement throughout my academic journey. Their belief in my abilities has been a constant source of motivation and inspiration, and I could not have completed this thesis without them.

Furthermore, I would like to express my gratitude to my colleagues, especially Ronaldo Vieira, who provided a welcoming and supportive environment in the lab, and whose conversations and insights contributed to the success of this research.

This work was partially supported by CAPES, CNPq, and Fapemig. Thank you all for your contributions, encouragement, and support.

"*We live on an island surrounded by a sea of ignorance.*
As our island of knowledge grows, so does the shore of our ignorance."
(John Archibald Wheeler)

Resumo

Jogos de estratégia em tempo real (RTS) apresentam um desafio único para agentes autônomos devido à combinação de vários problemas fundamentais de IA. Embora o Aprendizado por Reforço Profundo (Deep Reinforcement Learning - DRL) tenha demonstrado potencial no desenvolvimento de agentes autônomos para o gênero, as arquiteturas existentes muitas vezes apresentam dificuldades para se adaptarem a jogos que contenham mapas de dimensões variadas. Essa limitação prejudica a capacidade do agente de generalizar suas estratégias aprendidas a diferentes cenários.

Esta dissertação propõe uma abordagem inovadora que supera esse problema ao incorporar Spatial Pyramid Pooling (SPP) em um framework de DRL. Utilizamos a arquitetura encoder-decoder da rede GridNet e integramos a ela uma camada SPP na rede critic do algoritmo Proximal Policy Optimization (PPO). A camada SPP gera dinamicamente uma representação padronizada do estado do jogo, independentemente do tamanho inicial da observação. Isso permite que o agente adapte seu processo de tomada de decisão a qualquer configuração de mapa.

Nossas avaliações indicam que o método proposto melhora significativamente a flexibilidade e a eficiência do modelo no treinamento de agentes para diversos cenários de jogos RTS, embora com algumas limitações discerníveis quando aplicado a mapas muito pequenos. Embora sejam necessarios mais experimentos para consolidar essas descobertas, essa abordagem abre caminho para agentes de IA mais robustos e adaptáveis, capazes de se destacar em problemas de decisão sequencial com observações de tamanhos variáveis.

Palavras-chave: jogos digitais; estratégia em tempo real; aprendizado por reforço; aprendizado profundo.

Abstract

Real-time strategy (RTS) games present a unique challenge for autonomous agents due to the combination of several fundamental AI problems. While Deep Reinforcement Learning (DRL) has shown promise in the development of autonomous agents for the genre, existing architectures often struggle with games featuring maps of varying dimensions. This limitation hinders the agent's ability to generalize its learned strategies across different scenarios.

This thesis proposes a novel approach that overcomes this problem by incorporating Spatial Pyramid Pooling (SPP) within a DRL framework. We leverage the GridNet architecture's encoder-decoder structure and integrate an SPP layer into the critic network of the Proximal Policy Optimization (PPO) algorithm. The SPP layer dynamically generates a standardized representation of the game state, regardless of the initial observation size. This allows the agent to effectively adapt its decision-making process to any map configuration.

Our evaluations indicate that the proposed method enhances the model's flexibility and efficiency in training agents for various RTS game scenarios, albeit with some discernible limitations when applied to very small maps. While additional experimentation is needed to consolidate these findings, this approach paves the way for more robust and adaptable AI agents capable of excelling in sequential decision problems with variable-size observations.

Keywords: computer games; real-time strategy; reinforcement learning; deep learning.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Games are fundamental to human culture, serving as one of the oldest forms of social interaction. They enable people to expand their imagination and engage with others on cultural, educational, and social levels. Playing games allows individuals to develop their physical, intellectual, emotional, and creative potential. For these reasons, parents, educators, and teachers often use games as tools for children development and learning. Additionally, due to their high complexity, they present significant challenges to players and researchers alike, making them excellent benchmarking tools for evaluating various techniques, particularly those in the field of Artificial Intelligence (AI). Therefore, games are not only a vital aspect of human culture but also an important tool for personal and technological development.

The intersection of games and AI has witnessed remarkable advancements in recent years, offering a rich landscape for exploration and innovation. Among the various genres of games, real-time strategy (RTS) games stand out as a particularly fertile ground for research and development [Ontanón et al., 2013, Vinyals et al., 2017]. RTS games require players to make rapid decisions, manage resources, plan strategies, and adapt to dynamic environments, all in real time. This demanding nature of RTS games captivates players and presents a unique challenge for AI systems.

Reinforcement learning (RL), a subfield of AI, has emerged as a powerful paradigm for training agents to make sequential decisions in dynamic and uncertain environments. RL agents learn from interactions with the environment, aiming to maximize a cumulative reward signal [Sutton and Barto, 2018]. Unlike traditional rule-based AI, RL agents learn from experience, adapting their strategies through trial and error, much like human players do. This adaptability and learning capacity make RL particularly suited for RTS games' complex and dynamic nature.

Reinforcement Learning has garnered substantial attention in the field of artificial intelligence and machine learning in the last few years. The surge in interest can be linked to RL's remarkable successes in tackling complex, real-world problems, from mastering video games – such as Dota 2 [Berner et al., 2019] and Starcraft II [Vinyals et al., 2019] – to controlling vehicles [Wang et al., 2019] and optimizing supply chains [Rolf et al., 2023]. A crucial driver behind its rise is the integration of RL with deep neural networks,

known as Deep Reinforcement Learning (DRL) [François-Lavet et al., 2018], which has enabled the handling of high-dimensional data and empowered applications in computer vision, natural language processing, and robotics. Advances in hardware and algorithmic innovations have also made RL more efficient and practical, while the abundance of data from various sources further fuels its growth.

The application of RL techniques to RTS games has the potential to create intelligent game-playing agents that can rival and even surpass human performance, but there are some challenges that should be tackled for the widespread use of DRL in games. For instance, when the input scale changes, many advanced RL agents for modern games – such as the AlphaStar [Vinyals et al., 2019] – require extensive retraining to adapt to new conditions, a time-consuming and resource-intensive process. Specifically, in this work, we investigate current limitations of RL agents regarding input scale and propose a novel scale-invariant architecture that can work with several input sizes without requiring new training sessions. This model was created by combining and adapting well-known RL algorithms with computer vision techniques used in image segmentation tasks. We demonstrate that our proposed method improves the models' flexibility and provides a more effective and efficient solution for training autonomous agents in multiple RTS game scenarios as well as allowing for easier transfer learning between similar environments.

1.1 Motivation

The motivation behind this master's thesis stems from the key implications that RL can have on not only advancing the capabilities of AI agents in gaming but also on addressing broader real-world problems that demand intelligent decision-making in dynamic and uncertain environments.

Real-time strategy games have long served as benchmarks for testing AI capabilities due to their intricate and ever-changing nature. These games demand not only strategic foresight but also the ability to adapt rapidly to new information and opponents' actions. Traditional rule-based AI approaches, while effective to some extent, often perform poorly compared to human players in such dynamic environments. Reinforcement learning, on the other hand, offers a promising avenue to bridge this gap. By allowing an agent to learn from its interactions with the environment and iteratively improve its decision-making skills, RL can enable agents to acquire more adaptive and nuanced strategies, ultimately achieving gameplay levels that were previously unattainable [Vinyals et al., 2019].

Moreover, the practical applications of RL in RTS games extend far beyond the

gaming industry. The ability to develop agents that can excel in dynamic, competitive environments has relevance in many fields where decision-making under uncertainty is paramount. For instance, in autonomous robotics, RL-trained robots could make more informed decisions in complex, unpredictable environments, such as disaster response scenarios, where quick adaptations and efficient resource allocation are critical. In finance, RL algorithms could enhance portfolio management by optimizing investment decisions in rapidly changing markets. Additionally, RL-driven support systems could aid in treatment planning and optimizing patient care by adapting to evolving medical data and patient conditions. In essence, the insights gained from applying RL to the intricate challenges of RTS gaming can pave the way for more adaptive, intelligent, and efficient decision-making systems across various industries, fundamentally altering how we approach complex decision-making problems in the real world.

1.2 Objectives and Contributions

This thesis aims to advance the understanding and application of scale-invariant techniques in reinforcement learning within RTS games. Our main goal is to develop a scale-invariant agent for RTS environments. The primary contributions are as follows:

A Scale-Invariant Reinforcement Learning Architecture: We introduce a novel reinforcement learning architecture designed to enable efficient decision-making regardless of the input size and complexity of the environment.

Enhanced Reinforcement Learning Algorithms: We refine existing reinforcement learning algorithms to harness the capabilities of our scale-invariant model, integrating multiple environments with varying scales into a unified training routine. This approach results in more robust training and better generalization. We demonstrate the effectiveness of this technique using the Proximal Policy Optimization (PPO) algorithm, but it can be applied to others.

Autonomous Real-Time Strategy Game Agent: By integrating our novel architecture and improved PPO algorithm, we developed an autonomous agent for the μRTS framework that adapts to distinct map scales.

Besides these primary contributions, we have made other smaller yet significant contributions, including an adapted version of OpenAI's Frozen Lake, where the agent can fully observe the environment, and a scale-invariant autonomous agent for this mod-

ified environment. Furthermore, all the code used in this thesis has been made publicly available.

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 1 provides an introduction to the research topic, outlining the significance of games in human culture, and the relevance of RTS games in AI research. It also discusses the motivation, research goals, and contributions of this thesis.

Chapter 2 offers a comprehensive background, including the development of AI in games, fundamental concepts of reinforcement learning and its evolution, and RL in the context of RTS games.

Chapter 3 explores existing research and studies related to the specific topic of reinforcement learning in real-time strategy games. This chapter provides an in-depth analysis of prior work, identifying gaps and establishing a foundation for the current research.

Chapter 4 details the research methodology employed to address the research questions. It outlines the research design, techniques, and any tools used for the analysis. This chapter also discusses the rationale behind choosing specific methodologies and their suitability for the research objectives.

Chapter 5 presents the evaluation and findings of the research. It analyzes the data collected and discuss the results in the context of the research questions. This chapter also addresses any challenges faced during the research process and provide interpretations of the results.

Chapter 6 concludes the thesis, summarizing the key contributions, highlighting the limitations, and suggesting directions for future research in the topic of scale-invariant reinforcement learning in real-time strategy games.

Chapter 2

Background

In this chapter, we introduce the foundational concepts and methodologies required for this thesis. We begin by presenting the fundamentals of RL, detailing common approaches and their limitations. Building upon this foundation, we delve into the concept of deep learning and its pivotal role in enhancing RL through the emergence of deep reinforcement learning (DRL). Our discussion then extends to RTS games and their unique challenges. Furthermore, we highlight the significance of OpenAI Gym as a pivotal tool for RL research.

2.1 Reinforcement Learning

Machine Learning (ML) is a field of Artificial Intelligence that studies algorithms designed to learn from data. It encompasses three main approaches: supervised learning, unsupervised learning, and reinforcement learning. Reinforcement Learning (RL) is a computational approach to understanding and automating goal-directed learning and decision-making. Unlike other ML approaches, RL emphasizes learning from direct interaction with the environment without the need for supervision or complete models of the environment [Sutton and Barto, 2018]. This approach is particularly useful in situations where an agent needs to make sequential decisions on uncertain or dynamic environments, such as robotics and games. Figure 2.1 shows a basic RL diagram.

This thesis delves into the fundamental setting of single-agent reinforcement learning and examines three distinct methods for learning the correct agent behavior: value-based, policy-based, and actor-critic. By exploring these methods, we aim to gain a comprehensive understanding of the advantages and limitations of each approach, and their practical applications.

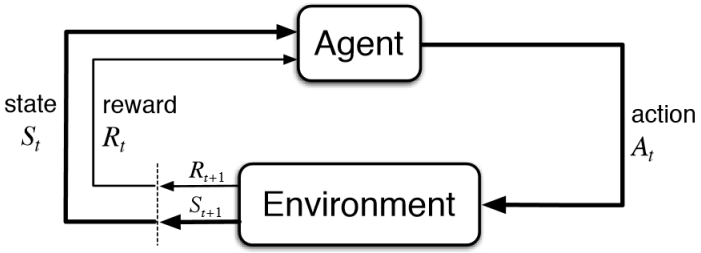

Figure 2.1: Reinforcement Learning diagram (Adapted from Sutton and Barto [2018]).

2.1.1 Markov Decision Processes

The cornerstone reinforcement learning problem is single-agent reinforcement learning, where only a single agent interacts with the environment. This problem is typically modeled by a Markov Decision Process (MDP), a mathematical framework for modeling sequential decision-making where the outcome at each stage depends mainly on the state of the environment and the agent's chosen action. An MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

$\mathcal{S}$ is the finite set of all possible environment states (also called state space). A state $s_t \in \mathcal{S}$ is a complete description of the environment at time t.

$\mathcal{A}$ is the finite set of all actions available to the agent.

$\mathcal{P}$ is the state transition probability function. It provides the probability of transitioning to state s' given that we are currently on state s and perform action a. Written as $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$.

$\mathcal{R}$ is the reward function. It determines the reward signal the agent will receive by performing action a in state s, and is used to formalize the goal of the agent. Written as $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$.

γ is the discount factor: a parameter in the interval $\gamma \in [0, 1]$ that denotes how much a future reward is valued at the current moment.

Ultimately, to be called an MDP, a system that follows this convention must obey the **Markov Property**, which specifies that the system's transitions must depend solely on the most recent state and action, without any prior history:

$$\mathcal{P}^a_{s_t s_{t+1}} = \mathbb{P}[s_{t+1} | s_t, a_t] = \mathbb{P}[s_{t+1} | s_0, a_0, s_1, a_1, \cdots, s_{t-1}, a_{t-1}, s_t, a_t]$$

Before addressing RL goals, we need to define a few other concepts: policies, trajectories, and return. A policy π is a distribution over actions given states that completely

defines the agent's behavior: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$. In deterministic policies, there is only one action with a probability of 1 for each state. In stochastic policies, multiple actions may have non-zero probabilities, and the agent's choice of action is probabilistic. The choice between a deterministic or stochastic policy depends on the problem and the agent's requirements. Stochastic policies are particularly beneficial for exploration, as they enable the agent to sample a variety of actions from the probability distribution, thus enhancing the learning process. Additionally, stochastic policies are also important when working with non-stationary MDPs, where the transition probabilities or the reward functions change over time. In these scenarios, the policies must adapt to account for the changing dynamics and rewards, in which case a stochastic policy can more easily adjust by gradually increasing the probabilities of actions that appear more promising.

A trajectory τ is a sequence of states, actions, and rewards that an agent experiences as it interacts with an environment over a period of time. A trajectory is often represented as a sequence $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \dots)$. The trajectory starts with an initial state s_0, followed by the action a_0 taken in that state, the reward r_1 received, and the new state s_1 that the agent transitions to, and so on. This sequence continues until the episode terminates, which can occur for various reasons, such as reaching a terminal state or after a fixed number of time steps.

The return $\mathcal{G}$ is the cumulative sum of discounted rewards that an agent receives along a specific trajectory: $G(\tau) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. The return is a way to quantify the total value or utility of a particular trajectory or sequence of actions taken by the agent in the environment. By analyzing the return obtained from trajectories following a policy, one can gauge how well the policy is performing and make informed decisions about refining or replacing the policy. This is the fundamental idea behind RL algorithms.

Reinforcement Learning aims to learn a policy that maximizes the expected return when the agent acts in accordance to it. This target policy is called *optimal policy* and might not be unique in some settings. As we will see in Section 2.1.2, there are three main approaches to approximating the optimal policy: value-based, policy-based, and actor-critic.

While the Markov Decision Process framework is a powerful tool for modeling sequential decision-making problems in a wide range of applications, it also has limitations. Two major challenges that arise in single-agent reinforcement learning are the curse of dimensionality and the exploration-exploitation trade-off.

The curse of dimensionality surfaces when the number of states or actions in a reinforcement learning problem increases exponentially with the number of individual variables or factors that collectively define these spaces (referred to as dimensions). As the number of dimensions grows, the amount of data needed to accurately estimate the value function or policy of the agent also grows exponentially, making it difficult to find

optimal solutions. To mitigate this challenge, researchers have developed techniques such as function approximation, state aggregation, and deep reinforcement learning.

The exploration-exploitation trade-off is a fundamental challenge in reinforcement learning that involves balancing the exploration of unknown states and actions with the exploitation of the agent's current knowledge to maximize reward. If the agent only exploits its current knowledge, it may miss out on discovering better strategies. On the other hand, if the agent only explores, it may waste time in states with low rewards. Researchers have developed various exploration strategies, such as epsilon-greedy, Upper Confidence Bound (UCB), and Thompson sampling, to balance exploration and exploitation [Sutton and Barto, 2018, Chapter 2].

The MDP framework can be extended to accommodate systems in which there are multiple agents interacting with the environment and with each other. This generalization is known as a **Markov Game** (also called Stochastic Game), defined by the tuple $(\mathcal{S}, \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{P}, \mathcal{R}_1, \ldots, \mathcal{R}_n, \gamma)$. This definition closely resembles that of an MDP, but with distinct sets of actions $\mathcal{A}_i$ for each agent i participating in the system. Similarly, each agent has its own reward function $\mathcal{R}_i$.

While this thesis mainly focuses on RTS games, a multi-agent system where multiple players compete against each other, we do not approach this problem as a multi-agent RL scenario. Instead, we model the opponents as components of the environment, resulting in a dynamic, non-stationary MDP. Consequently, we do not delve into the fundamentals and specific techniques of multi-agent RL in this context.

2.1.2 Reinforcement Learning Algorithms

RL algorithms can be divided into two major categories: model-based and model-free. Model-based algorithms rely on having access to (or learning) a model of the environment, including the transition probability function and the associated rewards. The main advantage of model-based algorithms is their ability to provide better sample efficiency, as they can use the model to simulate experiences without actually having to interact with the environment. However, in practice, a model of the world is usually not readily available or would be too complex to implement. In such cases, model-free algorithms become an interesting alternative that learns directly from experience through trial and error. Typically, model-free algorithms are also simpler and more scalable, making them a popular choice in many real-world applications of RL. This thesis will address only model-free algorithms since they offer a practical and versatile approach to reinforcement learning that is well-suited to a wide range of real-world problems where modeling the

environment is challenging or impractical.

A key distinction is also made between on-policy and off-policy algorithms, each representing a unique approach to learning optimal policies. On-policy methods directly evaluate and improve the policy used for decision-making. They learn action values for a policy that is near-optimal while still exploring. A prime example of an on-policy algorithm is State-Action-Reward-State-Action (SARSA) [Rummery and Niranjan, 1994]. Off-policy methods, in contrast, involve two policies: a target policy they aim to improve and a separate behavior policy used for exploration. The behavior policy can be more exploratory than the target policy, allowing the agent to explore different actions or strategies that might lead to better long-term rewards for the target policy, even if they are suboptimal in the short term [Sutton and Barto, 2018, Chapters 5, 11].

The first class of model-free algorithms we will address are value-based algorithms. These algorithms are built upon the concept that each environment state s or state-action pair (s, a) has a value associated with it: the expected return if you start at that point and follow a particular policy afterward. These methods are designed to approximate the value functions $V(s)$ and $Q(s, a)$, which will then be used to guide the policy an agent must follow to maximize its return. The Q-Learning algorithm [Watkins and Dayan, 1992] is one notable example of this class. It is an off-policy algorithm that learns the optimal Q-value function and is usually employed with an ε-greedy strategy, which selects a random action with probability ε and the action with the highest Q-value with probability $1 - \varepsilon$. Value-based approaches focus on learning the $Q(s, a)$ function and approximating the optimal policy by following a greedy strategy that selects actions based on the estimated $Q(s, a)$ values.

The second class of model-free algorithms we will go over are policy-based algorithms. In contrast to value-based, policy-based algorithms directly search the policy space. They utilize a parameterized policy $\pi_\theta(a|s)$ – usually in the form of a Neural Network – that is directly optimized by defining an objective function and using gradient ascent to reach an optimal point. As explained previously, we wish to maximize the expected return $\mathbb{E}[G(\tau)]$, which will be used to calculate the parameter's updates by deriving the expression $\nabla_\theta \mathbb{E}[G(\tau)] = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) G(\tau)]$, known as the policy gradient theorem. A notable example of a policy-based algorithm is REINFORCE [Williams, 1992], which uses Monte Carlo sampling to estimate the expected reward of each action and update the policy parameters accordingly. Policy-based algorithms are particularly useful in situations where the optimal policy is complex and difficult to derive from a value function. However, they can suffer from high variance and require careful tuning of hyperparameters to ensure stable convergence.

The last class of model-free algorithms we will cover are the Actor-Critic algorithms. These are hybrid algorithms that combine both value-based and policy-based models by using two separate estimators. The first agent – called critic – uses a value-

based approach to estimate the value function, while the second agent – called actor – uses a policy-based approach to represent the parameterized policy. During training, the critic provides feedback to the actor by estimating the value function of the current policy. The actor then adjusts the policy parameters based on this feedback to improve performance. This two-agent approach allows for a more stable and efficient learning process compared to using a single policy-based or value-based approach. Actor-critic algorithms have been shown to exhibit lower variance in gradient estimates compared to policy-based algorithms, which can be useful for training in complex environments. Some examples of actor-critic algorithms include Asynchronous Advantage Actor-Critic (A3C) [Mnih et al., 2016], Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015], and Proximal Policy Optimization (PPO) [Schulman et al., 2017].

2.2 Deep Learning

The evolution of Artificial Neural Networks (ANNs), also known as Neural Networks (NNs), has significantly improved the ability of machine learning models to learn complex patterns and make accurate predictions across various domains. Recent advancements in hardware have enabled Neural Networks to process large amounts of data with unprecedented efficiency, allowing them to excel in complex and unstructured tasks such as image and text analysis. This progress has led to the emergence of Deep Learning (DL), which refers to the capability of NNs to learn and represent increasingly complex and abstract concepts through the use of multi-layered networks.

2.2.1 Neural Networks

Neural networks are a core component of deep learning, drawing inspiration from the intricate workings of the animal central nervous system. The concept of an artificial neural network was first introduced through a supervised learning algorithm known as the Perceptron [McCulloch and Pitts, 1943, Rosenblatt, 1957, 1958], which consists of a single artificial neuron. The Perceptron is a binary classifier, a function that decides whether an element belongs to a class based on its features. It receives n feature values $X = \{x_1, x_2, x_3, \ldots, x_n\}$ and outputs a prediction $\hat{y} \in \{-1, 1\}$, indicating which class the element belongs to. Inside the Perceptron, we find n weights $W = \{w_1, w_2, w_3, \ldots, w_n\}$

and a bias b that are used to determine the element's class. The prediction $\hat{y}$ is calculated by performing a weighted sum of the input features with their respective weights, adding the bias, and passing the result to an activation function φ. This computation is written as $\hat{y} = \varphi(w^T x + b)$. Figure 2.2 shows the typical structure of an artificial neuron such as the Perceptron.

The activation function φ of the original Perceptron is the step function defined as:

$$f(x) = \begin{cases} -1, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{2.1}$$

However, this activation function fell out of favor in later architectures due to the emergence of better alternatives, such as the sigmoid function or the Rectified Linear Unit (ReLU). These alternative activation functions offer significant benefits over the step function, such as continuous differentiability, improved convergence properties, and the ability to model non-linear relationships between input and output. As a result, they have become the go-to choice for many neural network architectures.

Training the Perceptron is a simple procedure that requires a labeled dataset $D = \{(X^1, y^1), (X^2, y^2), \ldots, (X^k, y^k)\}$ containing k samples of elements and their respective classes. The samples are passed to the Perceptron individually, and its prediction is compared to the ground truth (the label). The internal weights are then updated by following the rules:

$$\begin{aligned} w_i &\leftarrow w_i + \alpha \cdot w_i \cdot x_i \cdot (y - \hat{y}) \\ b &\leftarrow b + \alpha \cdot b \cdot (y - \hat{y}) \end{aligned} \tag{2.2}$$

where α is the learning rate hyperparameter, used to control the magnitude of the updates and, consequently, the learning speed.

Despite its simplicity, the Perceptron served as a crucial building block for constructing more complex neural networks. Initially, it was received as a promising learning device, but it had various limitations, as demonstrated by Minsky and Papert [Minsky and Papert, 2017]. The most significant drawback was its inability to classify elements in non-linearly separable classes. A solution to this problem came with the Multi-layer Perceptron (MLP) and the Backpropagation algorithm [Rumelhart et al., 1986], which paved the way for significant advancements in the field.

As the name suggests, a MLP is an aggregation of several Perceptrons in multiple layers, where each layer's output is passed as input for the Perceptrons of the next layer. This way, each layer generates a new representation of the previous one. Every MLP must have at least one hidden layer, a layer between the input and output layers. The layer sizes can be arbitrarily big or small, with all layers containing the same number of neurons or each layer containing a unique quantity. Figure 2.3 shows a typical structure of a MLP network. MLPs are universal approximators, they can approximate any continuous

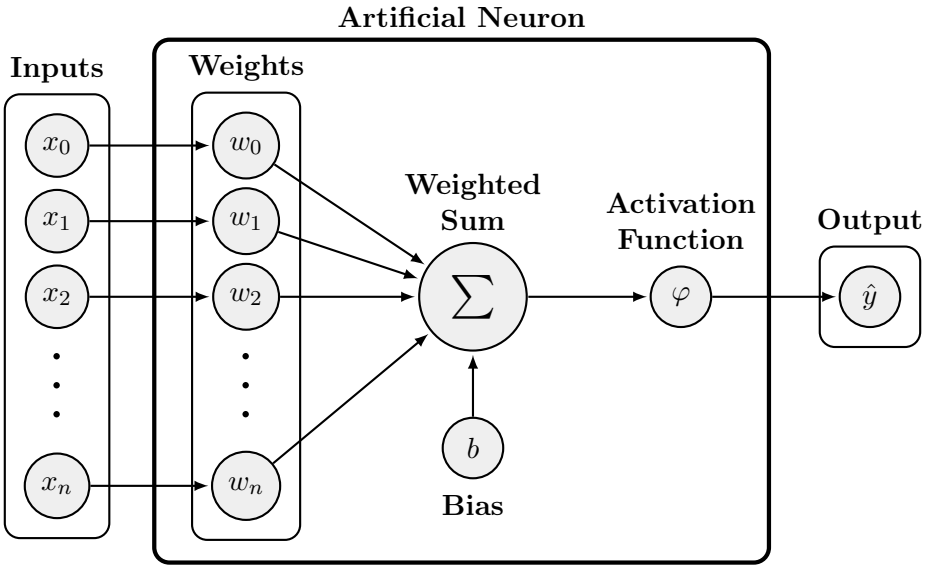

Figure 2.2: Typical artificial neuron's structure.

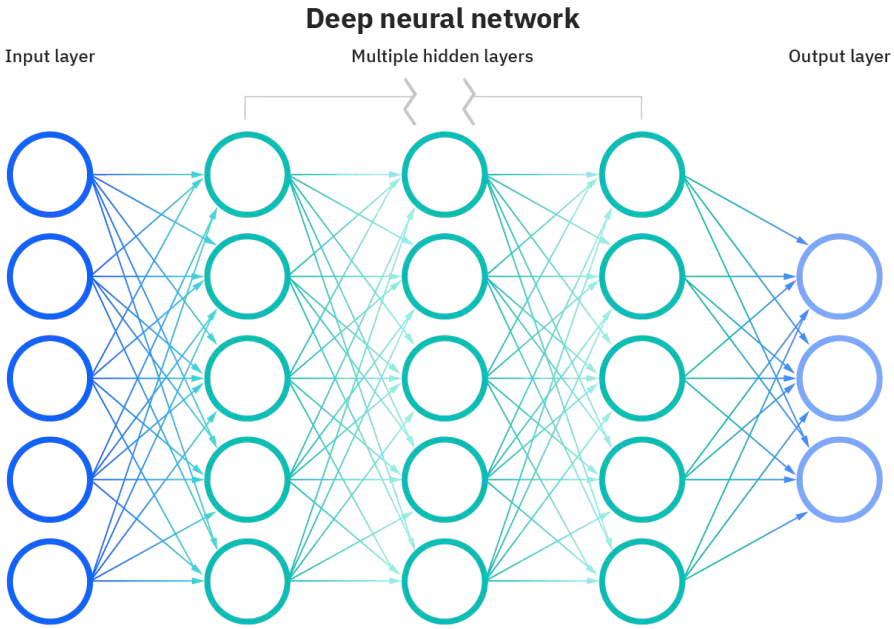

Figure 2.3: Multi-layer Perceptron Network Structure (Adapted from IBM's web page *What are Neural Networks?).* [1]

function with finite support [Hornik, 1991, Pinkus, 1999]. Therefore, they overcome the Perceptron's limitations regarding linearly separable classes.

Many algorithms can be used to train Neural Networks, the most popular being Gradient Descent (GD) [Ruder, 2016]. GD is an optimization algorithm that minimizes a parameterized objective function $J(\theta)$ by iteratively moving toward the steepest de-

[1] Available at: <https://www.ibm.com/cloud/learn/neural-networks>Accessed in: 2022-10-03

scent defined by the negative of the gradient $\nabla_\theta J(\theta)$ with reference to the parameters θ. Training an artificial network implies we are looking to improve its predictions over time. To evaluate the model, we utilize a performance metric called *loss function* that calculates the distance between the network's predictions and the ground truth values. This loss function is the objective function J we wish to minimize with GD. The Mean Square Error (MSE) and the Mean Absolute Error (MAE) are examples of loss functions commonly used to train NNs. The error is backpropagated throughout the network and used to compute the parameters' update for each neuron. As a result, the internal nodes of the network learn to capture important features of the domain, enabling more accurate predictions.

MLP networks are the foundation of Neural Networks, but there are many other architectures for Deep Learning, where some are better suited for specific problems. For example, Convolutional Neural Networks (CNNs) have revolutionized the computer vision field. They specialize in recognizing patterns in images, videos, and other spatial data. The core of CNNs are the convolutional layers, where a set of learnable kernels (also called filters) slides over the input image. Each kernel convolves across the width and height of the input, computing weighted averages of the input data at each spatial position according to the kernel's weights, producing a feature map. Figure 2.4 shows how a convolution is computed using a 2×2 kernel in a 3×4 input matrix.

The convolution process leverages *sparse interactions*, *parameter sharing*, and *equivariant representations*, all of which improve the performance of machine learning systems. In MLPs, each layer utilizes a weight matrix containing individual parameters that determine how each input element influences each output element. In contrast, convolutional layers utilizes kernels that are usually much smaller than the input volume, resulting in *sparse interactions*. For this reason, CNNs need to store and learn fewer parameters, reducing memory requirements and improving the efficiency of the model. Additionally, due to the smaller size of the kernels, convolutional layers apply the same set of weights (the kernel) across different regions of the input, which is known as *parameter sharing*. Instead of learning a unique set of parameters for each region of the input, convolutional layers only need to learn a single set that can be used across all regions, further increasing the efficiency of the model. The specific form of parameter sharing that convolutional layers use also confer the *equivariance translation* property to them, meaning that if the input is shifted, the output will be shifted in the same manner. This property is useful for tasks like image recognition, where the object might appear in different locations, or when processing time-series, where the convolution will generate a timeline showing when specific features occur in the input [Goodfellow et al., 2016, Chapter 9].

The output of convolutional layers typically undergoes a nonlinear function activation, just like the architectures discussed previously, but they are also usually followed by

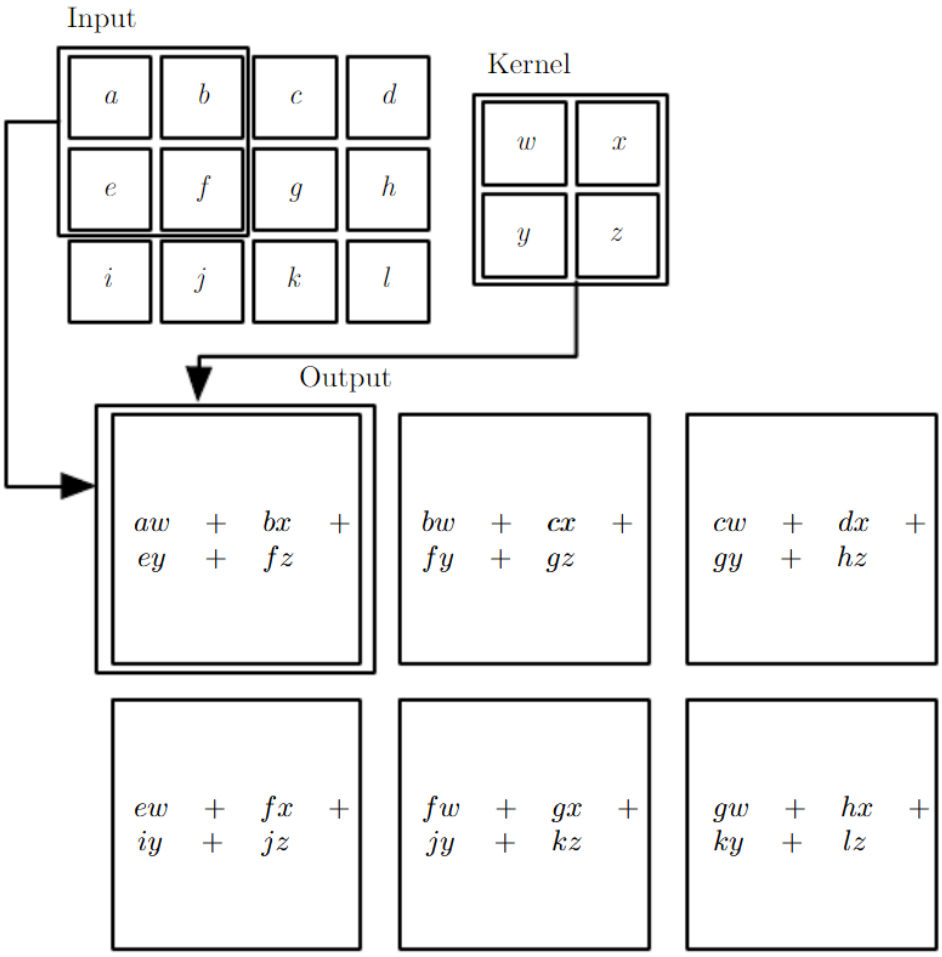

Figure 2.4: An example of 2-D convolution. The output is restricted to positions where the kernel lies entirely within the input data (Adapted from [Goodfellow et al., 2016, Chapter 9]).

a pooling layer. A pooling layer divides the input into small regions and performs a statistical aggregation of the data, such as taking the maximum or average value, within each region. This procedure produces representations that are invariant to small translations of the input, provided that most values within the pooling regions remain unchanged. This is particularly useful when we are interested in detecting the presence of a feature in a general region, rather than finding its exact location [Goodfellow et al., 2016, Chapter 9].

Pooling is also useful for handling inputs of varying size. For example, in classification tasks, CNNs typically include fully connected layers on the end of the network, which require a constant input size. By using a pooling layer and varying the size of its offset between pooling regions, we can produce a representation with constant dimension, regardless of the input size. For example, we can define a pooling layer that always divides the input into four quadrants and takes the maximum value of each quadrant, resulting in an output of four values that can be used by the fully connected layers to classify the

image. This is a specially important procedure for this thesis as we will use a pooling technique called Spatial Pyramid Pooling (SPP) to achieve a scale-invariant model, as we will explain in Chapter 4.

In summary, deep learning offers a variety of architectures that can be tailored to specific problem domains. By selecting an appropriate architecture and adjusting its parameters, deep learning models can achieve high accuracy and generalization performance in a wide range of complex applications.

2.2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is the combination of Deep Learning and Reinforcement Learning techniques. Its core objective is the same as traditional reinforcement learning: train agents to make sequential decisions based on the feedback received from their interactions with the environment. However, unlike traditional RL, which can struggle to handle high-dimensional and continuous state and action spaces, DRL uses deep neural networks to model and learn complex patterns within these spaces. The Deep Neural Netwroks (DNNs) empower RL agents to generalize more effectively and make efficient decisions, as similar scenarios are mapped to analogous values or policies. Consequently, DRL emerged as an invaluable approach for solving complex real-world problems.

The Deep Q-Network (DQN) [Mnih et al., 2013b, 2015] was the algorithm that marked the DRL ascension as a powerful and promising approach within RL. As discussed previously, in traditional RL, when the state and action spaces become large or continuous, the curse of dimensionality becomes a formidable challenge, making it exceedingly difficult to explore and represent the state-action space accurately. DQN overcame this challenge by utilizing deep neural networks as function approximators to approximate the Q-values of state-action pairs.

In DQN, a deep neural network estimates the Q-values, enabling the agent to effectively generalize across the state and action spaces, thus providing a scalable solution to high-dimensional and continuous problems. Two pivotal techniques for the success of DQN were experience replay and target networks. Experience replay, first studied by Lin [1992], involves storing the agent's experiences (i.e., state, action, reward, next state) in a replay buffer and then randomly sampling mini-batches from this buffer during training. This approach decouples the temporal correlation between consecutive experiences and stabilizes training. On the other hand, target networks stabilize the Q-value target during the learning process. By periodically updating a target network with the parameters of

the online Q-network, DQN achieves more stable and reliable training.

DQN's success was initially demonstrated by solving a range of challenging Atari 2600 games, achieving human-level or superhuman performance in many of them. This marked a significant breakthrough and showcased the potential of DRL in handling complex real-world problems that involve high-dimensional sensory inputs and action spaces, as well as delayed rewards.

Since the introduction of DQN, the field of DRL has continued to advance rapidly. Various extensions and improvements to DQN have been proposed, including Double DQN [Van Hasselt et al., 2016], Dueling DQN [Wang et al., 2016], and Rainbow [Hessel et al., 2018], each tailored to address specific challenges and domains.

In addition to DQN advancements, other significant DRL algorithms have also emerged, such as the Trust Region Policy Optimization (TRPO). TRPO is a policy optimization algorithm that addresses the challenge of ensuring stable and reliable policy updates by constraining the policy changes to be within a trust region. TRPO updates policies by taking the largest step possible towards performance improvement, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of the Kullback–Leibler (KL) Divergence, a measure of relative entropy or difference in information represented by two distributions. By controlling the magnitude of policy updates, TRPO prevents catastrophic policy collapses during training, making it a safer and more efficient algorithm.

While TRPO offers stable policy updates by constraining the policy changes within a trust region defined by the KL-divergence, it does so at a considerable computational cost. The use of second-order optimization methods, such as computing the Hessian matrix or its approximations, can be computationally intensive, especially when dealing with deep neural network. Proximal Policy Optimization (PPO) [Schulman et al., 2017] emerged as an alternative approach to policy optimization that aims to address some of the computational challenges associated with TRPO while maintaining stable and efficient policy updates.

PPO introduces a family of algorithms, with PPO-Clip being the primary variant. This family of algorithms stands out for their simplicity, ease of implementation, scalability, and competitive performance in various reinforcement learning tasks. Unlike TRPO, PPO-Clip does not incorporate a KL-divergence term in its objective function and, notably, does not impose a hard constraint on policy updates. Instead, PPO-Clip relies on a simple yet effective technique: clipping the objective function. This clipping mechanism mitigates the risk of the new policy diverging significantly from the old policy while allowing for meaningful policy updates. These advancements have led to the successful application of DRL in a wide range of applications, solidifying DRL's status as a cutting-edge technology with enormous potential.

2.3 Real-Time Strategy Games

Real-Time Strategy (RTS) is an strategy-oriented genre of digital games characterized by simultaneous player progression, in contrast to the turn-based style of most strategy games. Games occur on a map where two or more players engage in a free-for-all mode (where every player competes individually) or in a team-based mode (where players cooperate in groups) to defeat their opponents. In general, RTSs are a match where players develop their empires and try to achieve hegemony over their opponents. Most RTS games include the following components [Ontanón et al., 2013]:

Economy system: At its core, RTS games revolve around the management of resources. Every game has its unique set of resources that players must gather to make progress. For instance, StarCraft features minerals, and vespene gas, while Age of Empires has food, wood, gold, and stone. Collecting these resources is critical to building structures, training units, and making technological advancements. Since these resources are limited and often found in specific locations on the map, players must compete with their opponents to secure them and make strategic decisions about where and when to invest.

Building management: Players are responsible for allocating, constructing, upgrading, and demolishing buildings as needed. These structures can have a wide range of functionality and are generally divided into civilian and military categories. Military building include defensive structures and production of combat units, while civilian buildings usually include housing, resource-gathering, and research structures. Each of these buildings improves the player's empire by supplying new options for the player, as they can provide defensive and offensive power or may enable the production of more advanced units.

Unit Management: Players are responsible for producing units that are vital for resource gathering, base defense, and offensive maneuvers against opponents. With each unit equipped with unique abilities and excelling in distinct aspects, players must decide which to produce based on their strategy and the opponent's strengths and weaknesses.

Technologies: Technologies are also a fundamental element present in most RTS games, offering research options to enhance the capabilities of their units, buildings, and overall gameplay. These upgrades may vary in type and scope, depending on the game and its mechanics. Some technologies might improve the damage or range of certain units, while others might provide defensive bonuses to buildings or allow the construction of new units altogether. They influence the gameplay by opening

up new strategies and tactics. Players must balance the cost and time required to research technologies against the potential benefits that they can provide.

Fog of War: Players do not have a full view of the world map and can only see what is within the radius of their buildings and units. This mechanic introduces uncertainties about other players' actions along with the uncertainty of the state of the map. Information is crucial at every stage of the game, and players must thoughtfully decide when and how to scout enemy-controlled lands to gather information.

RTS games present a substantial challenge for artificial intelligence due to their complex nature, which involves several fundamental AI problems, particularly resource management, decision-making under uncertainty, spatial and temporal reasoning, multi-agent collaboration, and real-time planning [Buro, 2003]. These challenges require AI systems to be capable of making strategic decisions in a dynamic and unpredictable environment while also taking into account the actions of multiple agents and adapting to changing circumstances in real time.

Blizzard's StarCraft, the most well-known RTS franchise worldwide, has gained significant attention in the field of AI in recent years. The game features multiple units with different abilities, various maps with unique layouts, and intricate gameplay mechanics. StarCraft's complexity and strategic decision-making required to play at a high level make it an excellent challenge for AI agents. As a result, it has become a popular testbed for researchers to evaluate and develop new machine learning algorithms [Vinyals et al., 2017]. Several competitions have been held to evaluate the performance of autonomous agents in StarCraft, including the AIIDE StarCraft AI Competition[2] and the CoG StarCraft AI Competition[3]. These competitions not only led to the development of innovative techniques and algorithms that have pushed the boundaries of AI research but also to insights that have translated into advancements in other fields, such as robotics and autonomous vehicles.

2.4 Real-Time Strategy Games Developed for Research

While some research groups have access to large-scale computing systems and can tackle complex RTS games like StarCraft, the majority do not have such resources at their

[2] https://www.cs.mun.ca/~dchurchill/starcraftaicomp/index.shtml
[3] https://cilab.gist.ac.kr/sc_competition/
[4] Available at: <https://starcraft2.com/en-us/media>Accessed in: 2022-10-05

Figure 2.5: StarCraft II screenshoot (Adapted from Blizzard's StarCraft II website[4]).

disposal. As a result, they often resort to using simpler RTS games developed for research purposes. These games offer a more controlled environment where researchers can focus on specific aspects of AI development without being overwhelmed by the complexity of larger-scale games. They can be used for experimentation and testing to provide valuable insights into the workings of AI, which may help develop new strategies and techniques that can be applied to more complex games in the future. A few examples o such games are: MiniRTS [Tian et al., 2017], Deep RTS [Andersen et al., 2018], and μRTS [Ontanón, 2013, Huang et al., 2021].

Minirts is a lightweight and flexible research platform that is part of the ELF project [Tian et al., 2017]. It is designed to be easy to use and modify, making it a good choice for researchers who want to experiment with different AI algorithms. Minirts is also very fast, capable of running at 40,000 frames per second on a single core. This makes it ideal for training reinforcement learning agents, which can require a large number of training iterations.

DeepRTS, on the other hand, is a more complex RTS game environment that is designed to be more challenging for AI agents. It features a larger map, more units, and more complex game mechanics. DeepRTS is also more visually appealing than Minirts, making it a good choice for researchers who want to create visually appealing demos of their AI agents. All three frameworks mentioned in this section are open-source projects, so researchers can freely modify and extend them to meet their needs. They are also compatible with a variety of AI frameworks, making it easy to integrate them into existing research projects.

μRTS is a lightweight open-source platform developed in Java. The game features

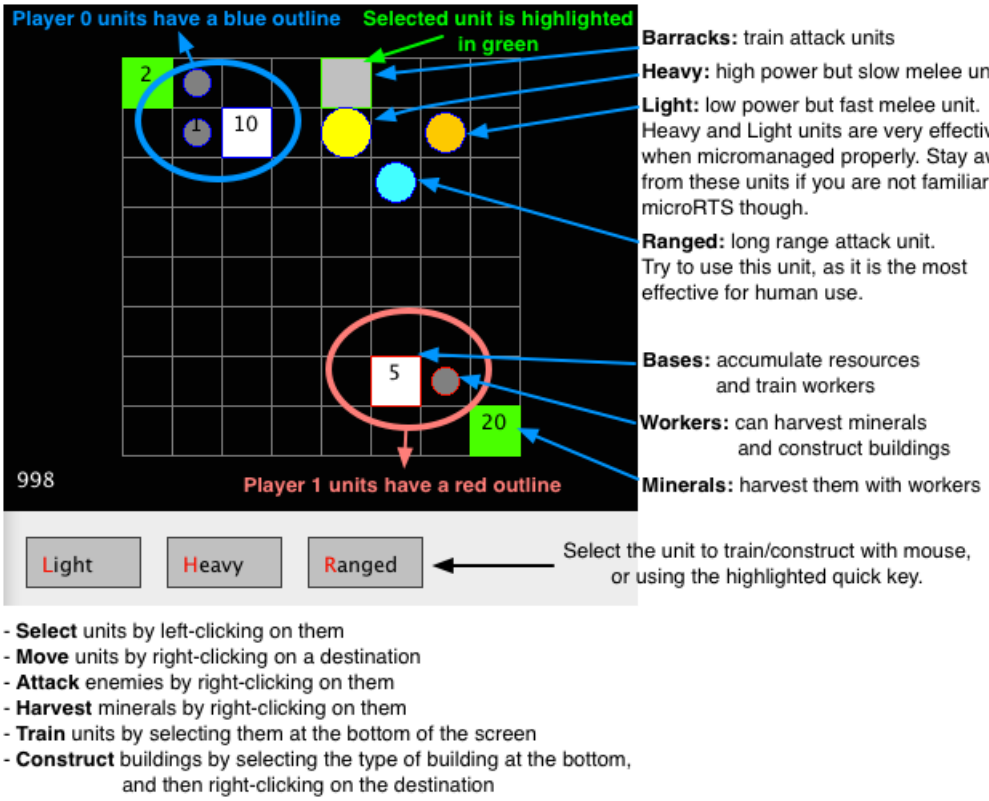

Figure 2.6: Image of a μRTS match in a 8×8 map showcasing the units, structures, and resources. Circles represent units, their type delineated by size and fill color, while squares denote various structures, each color-coded for easy identification (Adapted from the official μRTS GitHub repository).

a grid-based map and offers limited complexity compared to commercial RTS games while maintaining the core RTS challenges. In μRTS, each player begins the game with a base, where they can store resources and produce workers. The game features a single resource type: minerals, which are required for producing units and structures. Players have access to four unit types: workers, light, heavy, and ranged. Workers are versatile, low-cost units that gather resources and construct buildings but are generally weak in combat. The other three are specialized military units, each with distinct combat roles. Light units boast high speed and a reasonably strong attack power but lack durability. Heavy units excel in both attack power and durability but move slowly. Ranged units stand out for their ability to attack from a distance, while other units must engage enemies at close range. To produce these military units, players must first construct barracks structures. Figure 2.6 illustrates the units, structures, and user interface of a μRTS match.

The μRTS engine supports various game scenarios and allows for easy customization and adaptation to different research objectives. Its flexible interface allows seamless integration of AI agents, catering to both rule-based and machine learning-driven approaches. The framework also provides a suite of evaluation metrics to assess AI agent

performance, such as win rates, resource efficiency, and decision-making speed. These metrics aid in comparing and analyzing different AI strategies. For these reasons the μRTS framework has established itself as a valuable tool for advancing research in the domain of real-time strategy games and artificial intelligence. As a result, the μRTS AI Competition[5] was created, with the primary objective of motivating the development of AI for RTS games while minimizing the amount of engineering required.

2.5 OpenAI Gym

OpenAI Gym [Brockman et al., 2016] is a widely used open-source toolkit that provides a diverse collection of environments designed to benchmark and facilitate the development of RL algorithms. By offering a standardized interface and a broad range of tasks, including classic control problems, Atari games, and simulated robotics, OpenAI Gym enables researchers to systematically prototype, test, and compare the performance of their algorithms. Its ability to streamline the experimental process allows easy reproducibility and results sharing within the RL community. Moreover, OpenAI Gym's extensibility supports the creation of custom environments, further fostering innovation and the exploration of new RL applications. This toolkit has become an important tool to help advancing the state of the art in RL, as it provides the necessary infrastructure for rigorous experimentation and the development of more robust and generalizable RL solutions.

The Frozen Lake environment[6] is a classic grid world simulation in OpenAI Gym, designed to evaluate reinforcement learning algorithms. It represents a grid of frozen tiles, with the agent starting at a designated position and aiming to reach a goal tile without stepping on hazardous tiles. However, the catch is that the ice can be slippery, causing the agent to slide in unintended directions. Due to its inherent uncertainty, it forces algorithms to account for the probabilistic nature of state transitions, making it an effective testbed for evaluating RL agents. The environment comes with varying sizes and complexities, offering different levels of challenge for reinforcement learning agents. Additionally, the simplicity of the environment allows for clear visualization and interpretation of the learning process.

Each tile in the Frozen Lake environment is represented by one of four types: a start tile (S), a frozen tile (F), a hole (H), or the goal (G). The agent navigates through the grid by taking actions such as moving up, down, left, or right. The goal is for the

[5] <https://sites.google.com/site/micrortsaicompetition
[6] <https://gymnasium.farama.org/environments/toy_text/frozen_lake/

Figure 2.7: Screenshot of the Frozen Lake environment.

agent to learn an optimal policy to navigate from the starting position to the goal while avoiding the hazardous holes, making it a popular benchmark for testing reinforcement learning algorithms' ability to handle stochastic environments and long-term planning.

Overall, the Frozen Lake environment's blend of simplicity and complexity makes it a versatile and challenging setting for advancing the understanding and development of intelligent agents. In Figure 2.7, we can see a graphical representation of the Frozen Lake environment.

Chapter 3

Related Work

This chapter provides a comprehensive review of related literature. This review begins by offering a historical perspective of AI in games, tracing its evolution from basic rule-based systems to more modern techniques employed today. Following this historical context, we present a review of recent advancements, with particular emphasis on research directly relevant to the topic of this thesis.

3.1 Artificial Intelligence in Games

Since the inception of electronic computers, people have been intrigued by the possibility of machines matching or even surpassing human intelligence [Turing, 2009]. However, quantifying intelligence is a challenging task, and there has been much interest in finding ways to measure AI capabilities. One approach is to verify AI performance in assignments that are difficult for humans, such as classic strategy games like Chess or Go. These games have long been recognized as some of the most difficult tasks for humans, and as such, they have been used extensively in AI research [Yannakakis and Togelius, 2014].

The following sections provide a concise overview of how games have become essential prototyping and benchmarking tools for AI, exploring developments in both board and digital games. We will highlight key techniques and discuss notable accomplishments in the field, including Deep Blue's victory over world chess champion Garry Kasparov, AlphaGo's triumph over european champion Fan Hui and world champion Lee Sedol in Go, and AlphaStar reaching the top rank of Starcraft II.

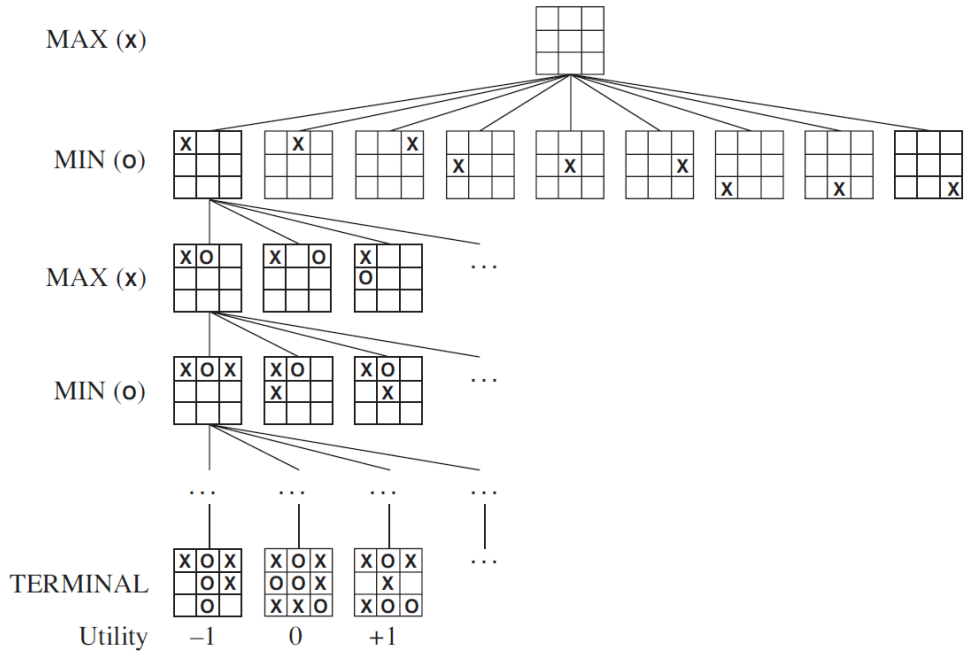

Figure 3.1: Partial minimax tree in a game of tic-tac-toe (Adapted from Russell [2010]).

3.1.1 Autonomous Agents for Classic Games

The early foundation for AI development in games was set in 1950 when Shannon [1950] presented the Minimax Algorithm (based on v. Neumann [1928] Minimax Theorem), a tree search algorithm designed to find the optimal action for a player assuming his opponent is also playing optimally. It examines the game tree, a tree where nodes are game states and edges are actions, to determine which move should be performed [Russell, 2010]. This is an effective procedure for simple games with small trees such as tic-tac-toe, as seen in Figure 3.1. However, a game tree can contain more states than there are atoms in the observable universe[1] in more complex cases, causing Shannon's algorithm to be impracticable due to the amount of memory and time it would require. Two common approaches to minimize this problem are *pruning* [Edwards and Hart, 1961], which cuts off branches that do not need to be searched in the game tree, and *cutoff with heuristic evaluation functions*, which estimate the value of a node without analyzing all of its children [Shannon, 1950]. Both techniques can significantly reduce the computation time of the algorithm. Pruning maintains an optimal solution, whereas cutoffs may lead to suboptimal solutions, depending on the heuristics used.

A few years later, IBM's engineer Samuel [1967] used a modified version of Minimax to develop his AI for Checkers, one of the first games tackled by researchers. The big dif-

[1]It is estimated that the observable universe has between 10^{78} and 10^{82} atoms, while games can have more than 10^{200} states [Burmeister and Wiles, 1995].

ference between his algorithm and Shannon's original Minimax is that Samuel added two learning mechanisms to his version. The first one, *Rote-learning*, allowed his program to record all states encountered during play and their associated results from computations. Although this is neither an advanced nor complex form of learning, it helped his agent reduce computation time in most situations. The second (and more important) learning procedure was a generalization learning in which the agent continuously updated the coefficients used to evaluate the board in look-ahead positions. This allowed his program to achieve better evaluation functions for non-terminal states over time.

Chess, one of the most popular strategy games in Western culture, could not be left out and has also been the subject of extensive research. First attempts at developing autonomous agents capable of masterfully playing chess date back to 1948 with Alan Turing's and David Champernowne's Turochamp [Copeland, 2004]. They designed the first algorithm for chess but could never finish its implementation since it was too complex for the computers at the time. In 1997 we had a great breakthrough in chess, with IBM's Deep Blue [Campbell et al., 2002] defeating then-reigning World Chess Champion Garry Kasparov in a six-game match. At the time, Deep Blue was considered a supercomputer, a massively parallel system designed for chess game tree searches. By using a hybrid hardware/software system with non-uniform searches, Deep Blue reached a searching capacity orders of magnitude higher than its predecessors.

Another interesting case is the game of Go, which first appeared circa 1000 – 400 BCE [Shotwell, 1994] and is believed to be the oldest board game still played today. It is one of the more challenging classic games due to its considerable branching factor, a value computed from the number of children at each node on the game tree that dictates the time complexity of searching the tree [Edelkamp and Korf, 1998]. To put it into perspective, Go has an average branching factor of 200, while chess has an average of 35 [Burmeister and Wiles, 1995]. Albeit Go has been studied since the 1960s [Zobrist, 1969, 1970], only in 2015 an AI defeated an expert human player on it. This was achieved by Deep Mind's AlphaGo [Silver et al., 2016], which combines Monte Carlo Tree Search (MCTS) with two deep neural networks that are used to estimate how good a board position is and which moves should be made. Professional levels of performance were only achieved in Go with the help of deep learning and reinforcement learning, which are also key to solving more complex games, as we will see in the following section.

3.1.2 Autonomous Agents for Modern Games

In the previous section, we have addressed only classic board strategy games, but modern games have been an important testbed for recent Deep Learning advancements. Digital games can be far more complex and challenging than classic board games. They can have immense space states and might require mastery of several independent skills. Although the classic algorithms discussed earlier can still be applied to these games, they often perform poorly due to their inability to handle the complexity and high-dimensional spaces typical of modern digital games.

More often than not, commercial games are published with some form of hard-coded AI opponents, with Finite State Machines (FSM) being the traditional approach for these agents. Although these opponents can be challenging for beginner human players and even achieve a significant amount of success, their performance falls off quickly against more experienced and adaptive players who can exploit strategy flaws [Pan and Yang, 2010].

This limitation of simpler AI systems in adapting to complex games and player strategies highlights the need for more advanced AI techniques that can learn and evolve. One notable advancement in this field happened through the Arcade Learning Environment (ALE) [Bellemare et al., 2013] – a framework developed to facilitate research and experimentation in the field of artificial intelligence on classic arcade games, based on the Atari 2600 video game console. In 2015, Mnih et al. [2015] achieved a significant breakthrough in reinforcement learning with the Deep-Q Network (DQN) by creating an autonomous agent that employed Deep Neural Networks (DNNs) with Reinforcement Learning. The agent surpassed a professional player in 29 out of 49 games tested, effectively marking the beginning of Deep Reinforcement Learning (DRL).

In recent years, Deep Reinforcement Learning has succeeded in many challenging games, such as Dota 2, a Multiplayer Online Battle Arena (MOBA) where two teams of five players compete against each other to destroy the enemy base. In 2019, OpenAI Five [Berner et al., 2019], a group of five artificial intelligence agents, reached a significant milestone by defeating the world champions in Dota 2. This achievement resulted from a 10-month training period using Deep Reinforcement Learning, where the agents engaged in self-play to improve their performance. OpenAI Five's victory demonstrated the potential of autonomous agents to exceed human performance in highly complex and dynamic environments.

Another remarkable example is StarCraft II, a popular Real-Time Strategy Game regarded as one of the most difficult professional esports. StarCraft II rates players in seven competitive ranks according to their skills and performance: Bronze, Silver, Gold, Platinum, Diamond, Master, and Grandmaster, with bronze being the lowest and

Grandmaster the highest. DeepMind's AlphaStar [Vinyals et al., 2019] employed a multi-agent reinforcement learning algorithm that uses human and agent games data to reach the Grandmaster rank, above 99.8% of officially ranked human players.

The development of an autonomous agent for a commercial RTS game is a highly challenging task, requiring extensive computational resources and specialized algorithms. AlphaStar stands out as the most successful attempt to date, which was possible mainly because of the employment of advanced deep reinforcement learning techniques. The training of AlphaStar required thousands of CPUs and GPUs for long sessions, making it a resource-intensive endeavor. Other teams have attempted similar approaches, but they have also relied on significant computational resources to achieve comparable results [Wang et al., 2021].

This thesis centers on the study of simpler RTS games, specifically on μRTS, a leading framework in the field for research purposes. Early efforts in developing autonomous agents for μRTS revolved around game-tree search [Ontanón, 2013], such as alpha-beta search [Churchill et al., 2012], Monte Carlo Tree Search (MCTS) [Churchill and Buro, 2013, Uriarte and Ontanón, 2014], or Puppet Search [Barriga et al., 2015, 2017]. The MCTS family of algorithms incrementally builds a partial game tree. Through repeated iterations, MCTS selects actions based on a trade-off between exploration and exploitation, and gradually expands the leafs of the current search tree, refining its understanding of the decision space. Each iteration involves four key steps: selection, expansion, simulation, and backpropagation. Selection involves navigating the tree and choosing which nodes to explore. Expansion adds new nodes to the tree, representing unexplored actions. Simulation involves running play-outs from these newly expanded nodes to estimate their potential outcomes. Finally, backpropagation updates the statistics of visited nodes based on the results of these simulations. Over many iterations, MCTS converges towards optimal or near-optimal decision-making strategies, making it a versatile technique used in many AI problems.

Although these approaches achieved significant performance in μRTS, they also exhibited notable limitations. Game-tree search methods are susceptible to high branching factors in decision trees, which can lead to exponential growth in the size of the search space, thereby increasing the computational requirements. Moreover, MCTS methods typically require a forward model of the game or a means of simulation, but they are not always readily available. Due to these inherent constraints, they gradually lost favor within the field.

Recently, the field witnessed promising advancements driven by Deep Reinforcement Learning algorithms such as Proximal Policy Optimization (PPO) [Schulman et al., 2017] and Deep Q-Network (DQN) [Mnih et al., 2013a], which have demonstrated strong performance across various domains. Within the μRTS framework, Huang et al. [2021] leveraged PPO alongside Grid-Wise Control and Invalid Action Masking [Huang and On-

tañón, 2022], resulting in an agent capable of efficiently outperforming state-of-the-art opponents available at the time. Despite its effectiveness, this model exhibits the usual lack of flexibility of DRL approaches: changes in map size require modifications to the network architecture and retraining.

3.2 Scale-Invariant Models in RTS Games

Deep Reinforcement Learning (DRL) typically struggles with handling varying input sizes due to the architectural constraints of neural networks commonly used in these systems. Most DRL algorithms rely on fixed-size input layers, which means they expect input data of a constant dimension. This requirement poses challenges when dealing with environments or tasks where the number of features or the structure of the input can change dynamically. For example, in scenarios like multi-agent systems, the number of agents can vary, altering the input size; or in games the players may experience maps of different sizes. Although input resizing methods such as cropping [Krizhevsky et al., 2017] or warping [Girshick et al., 2014] offer a workaround, they introduce a preprocessing step that can cause loss of information or distortions that can hinder the agent's learning.

Adapting to these variations requires specialized neural network architectures, such as those using attention mechanisms or graph neural networks, which can handle inputs of variable size by focusing on different parts of the input space or dynamically adjusting connections based on the input structure. Incorporating attention mechanisms into multi-agent control offers a promising avenue for managing diverse observation spaces. An instance of this is evident in the use of a Transformer-based approach [Vaswani et al., 2017] for multi-agent credit assignment and joint action evaluation within the context of the Starcraft Multi-Agent Challenge [Samvelyan et al., 2019].

An alternative approach to addressing the challenge of varying input sizes involves adapting the observation model. Using fixed-size representations may result in inefficient memory usage and processing resources in certain contexts. Given that much of the environment information is associated with entities, entity-based representations can offer notable efficiency gains, especially in sparsely populated domains. Graph Attention Networks (GATs) [Veličković et al., 2017] have been applied in single-agent control scenarios using an entity-based methodology within environments like the Arcade Learning Environment (ALE) [Bellemare et al., 2013] and Simple Playgrounds [Jankovics et al., 2022]. Moreover, GATs have been employed in Multi-Agent Reinforcement Learning settings to tackle Starcraft mini-games [Yun et al., 2021] using a decentralized approach. However, it is important to note that these applications were primarily tested on specific tasks rather

than entire Real-Time Strategy (RTS) matches.

Beyond these approaches, recent advances have introduced programmatic strategies as an effective alternative for RTS games. These strategies offer enhanced interpretability and computational efficiency compared to traditional search algorithms. For example, systems like LS2 [Marino et al., 2021] synthesize strategies through domain-specific languages and self-play optimization, outperforming search-based approaches and even strategies crafted by human programmers. Meanwhile, Sketch-SA [Medeiros et al., 2022] enhances the synthesis process by learning "sketches" from behavioral cloning, even from weak players, which accelerates the synthesis and enables the generation of robust strategies that defeated top competitors in recent μRTS tournaments. These studies demonstrate the potential of programmatic approaches to produce competitive, efficient, and explainable RTS game AI.

In our novel approach, we integrate Grid-Wise Control with Spatial Pyramid Pooling, creating a scale-invariant architecture tailored for sequential control problems. This innovation yields a versatile and efficient agent capable of seamlessly managing a varied number of units across distinct input sizes without requiring structural modifications. Leveraging insights from the work of Huang et al. [2021], particularly techniques like Invalid Action Masking and Action Composition, our approach is grounded on established methodologies. Additionally, we incorporate Reward Shaping [Mataric, 1994], accelerating the Reinforcement Learning process by introducing a series of small rewards tailored to guide the agent swiftly toward its ultimate objective.

Our proposed model introduces two significant enhancements over the literature. Firstly, we introduce a novel network architecture featuring Spatial Pyramid Pooling layers, producing a flexible scale-invariant network. Secondly, we implement a revamped training methodology that capitalizes on experiences garnered from maps with varying dimensions and representation sizes. This strategy approach generates a more comprehensive and robust learning process, enriching the agent's ability to tackle complex environments effectively.

Chapter 4

Methodology

This chapter outlines the methodology employed in implementing scale-invariant reinforcement learning techniques within RTS games. Initially, we discuss the challenges posed by scale-invariant learning in RTS games and examine various techniques that can effectively address individual challenges. Subsequently, we present our solution designed to overcome the majority of these challenges within a unified model. We highlight the flexibility of this novel model and showcase its capacity to operate seamlessly across different scales while enhancing its generalization abilities when used in appropriately tailored training routines.

4.1 Challenges of Scale-Invariant Learning

Despite all the success it has achieved recently, DRL still faces significant challenges, particularly when it comes to handling variable input sizes. DRL approaches often struggle and require retraining when working with different scales of input, even if the underlying environment has the same mechanics (e.g. training on the same game with different map sizes). Input resizing techniques alleviate the problem but require a preprocessing step and the definition of the neural network input size beforehand, which might not properly fit in unforeseen data (e.g. the predefined input is too small, and resizing a large input causes loss of information).

When it comes to mastering RTS games, we face multiple challenges regarding variable input sizes. The first one is due to the persistent and unpredictable fluctuations in the number of units under a player's control. This volatility stems from various factors such as the training of new units and the construction of additional buildings, as well as from casualties and destruction incurred during combat. The dynamic nature of these games makes it exceedingly difficult to foresee the precise quantity of units a player will have at their disposal at any given moment. Consequently, players must master the skill of micromanaging an ever-changing and often expansive array of units. Using a preprocessing

step to identify the entities in play would still imply in having variable input and output sizes, which can be a problem for most DRL approaches, since Neural Networks (NNs) usually require fixed-size input and output.

The second major challenge revolves around the game maps. Most RTS games offer a variety of maps, each presenting unique challenges and characteristics. Crucially, these maps can vary significantly in size. In games like μRTS, where the gameplay is grid-based, using the game map as input for the agent is a common approach in DRL. However, whenever the map size changes, the input size also changes, potentially disrupting the agent's functionality. Even using the game screen as input does not solve this issue. Screen resolution might vary based on the user's device, or users might adjust the resolution to enhance clarity and capture more game details, thereby potentially impacting the agent's performance. This challenge also extends to other domains, where sensors might be upgraded to offer higher resolution, demanding adaptive solutions to handle varying input sizes effectively. This section will go over techniques that can be employed to address these challenges and how they are used in different contexts.

4.1.1 Grid-Wise Control

As previously explained, controlling a variable number of agents poses a significant challenge in RL. To address this issue, Han et al. [2019] introduces an architecture that leverages spatial joint representation and grid-wise actions to effectively control an arbitrary number of agents. Each agent is controlled independently by assigning actions to each cell of the grid and performing actions based on the cell the agent occupies.

This method can be applied to any environment that can be divided into a grid, and is based on a convolutional encoder-decoder network. One could say it was inspired by the architectures commonly used in image segmentation tasks [Long et al., 2015], where the model has to assign semantic labels to each pixel in an image, but instead it, assigns actions to each cell in a grid. The encoder takes the state information as a grid feature map and encodes it into a latent representation. The decoder then decodes the latent representation into a grid of actions that will be executed by the agents.

Considering a grid with dimensions (w, h, c_s), where (w, h) represents the scale of the grid and c_s is the number of feature planes[1], GridNet takes a state $s \in \mathbb{R}^{w \times h \times c_s}$ as input and predicts an action a_{ij} for each grid position (i, j) with $1 \leq i \leq w$, $1 \leq j \leq h$.

[1]Feature planes are structured representations of the state in grid-based environments. They are typically 2D or 3D tensors, with each plane capturing a specific characteristic or aspect of the environment for every cell in the grid.

The resulting output is defined as an action map a with dimensions (w, h, c_a), where c_a indicates the action dimension of the agents. In Figure 4.1 we can see an illustrative example of a GridNet.

In RTS games, actions are usually defined by a combination of parameters, such as ordering a unit to build a base at position (x, y). In this case, the action type would be *build*, the structure type would be *base*, and the target location would comprise coordinates x and y. As a result, the dimension c_a must contain all the different parameters necessary to predict and compose any action and its specifications.

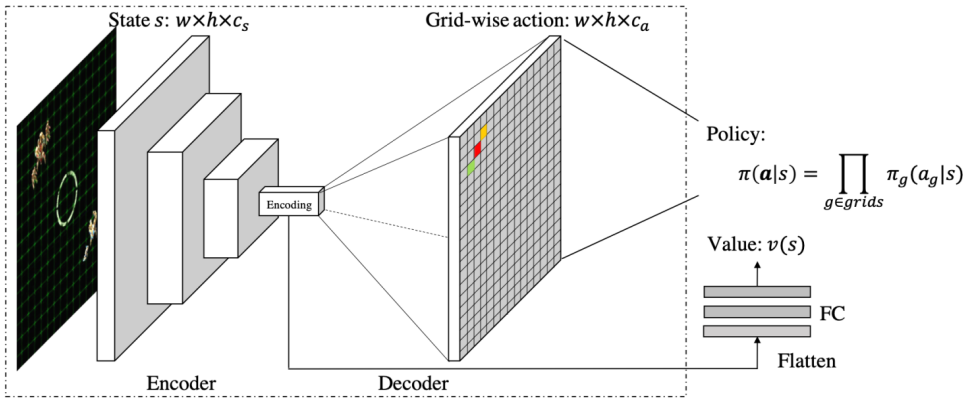

Figure 4.1: Example of a GridNet in the context of an actor-critic agent (Adapted from Han et al. [2019]).

4.1.2 Spatial Pyramid Pooling

Convolutional Neural Networks (CNNs) have revolutionized Deep Learning, especially in the computer vision field. Still, one significant limitation of many CNN architectures is their requirement for fixed-size input. This limitation is primarily attributed to CNN architectures typically ending with one or more fully connected layers, which demand a consistent input size to function properly. Each neuron in a fully connected layer is connected to every neuron in the previous layer. Consequently, the number of parameters in these layers is determined by the dimensions of the input. If the input size varies, the number of parameters in the fully connected layers would need to change accordingly. Nonetheless, real-world applications tend to encompass vastly heterogeneous data with images of varying sizes, which may be incompatible with CNNs by default. To circumvent this problem, many models employ preprocessing routines such as cropping [Krizhevsky et al., 2017] or warping [Girshick et al., 2014] to fit the images to the input size. While these techniques can make the images compatible with the network, they may result in

loss of information, especially in cases where crucial details are present at the image boundaries. As a result, they often reduce model accuracy and may even compromise recognition in some cases. Additionally, preprocessing steps can add complexity to the overall workflow.

A more effective alternative to this problem is the Spatial Pyramid Pooling (SPP) [He et al., 2015], a computer vision method that addresses the issue by generating a fixed-length representation regardless of input dimensions. By introducing a multi-level pooling technique over spatial cells, SPP handles input images of varying sizes without compromising on the network's performance. SPP addresses this limitation by dividing the input feature maps into a spatial grid and pooling features from each grid cell. Like most pooling techniques, it uses filters to generate a summarized representation from a given feature map. However, unlike most pooling, the filter size is variable and dependent on the input size, while a predefined grid dimension generate a constant-shape representation. As the name implies, SPP can also combine several filters, creating a pyramid-like structure. The result of each layer of the SPP is combined into a single vector that always has the same size, regardless of the input size. An example can be seen in Figure 4.2.

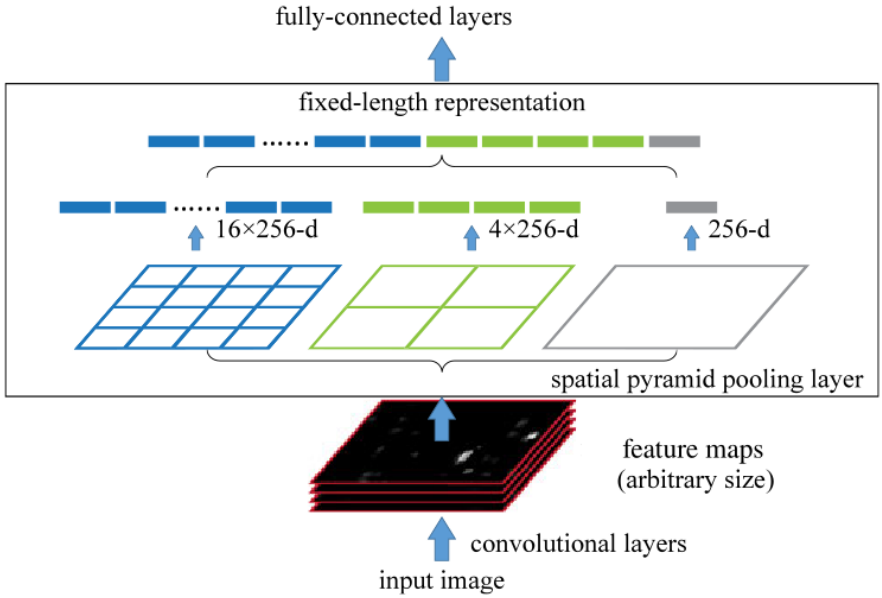

Figure 4.2: An example of Spatial Pyramid Pooling, assuming that the latest convolution layer yields 256 feature maps. The first SPP level has 16 bins of dimension 256, the second level has 4 bins and the third level has 1 bin. The final representation given by SPP has length $(16 + 4 + 1) \times 256$ regardless of the input (Adapted from He et al. [2015]).

4.2 Scale-Invariant Model

To address the limitations associated with fixed input sizes in current RL models, we propose a novel solution that integrates techniques from Grid-Wise control and SPP. This approach aims to create scale-invariant reinforcement learning agents, offering a more versatile and adaptable framework. This research culminated in the development of two distinct agents. The initial agent – detailed in Section 4.2.1 – was designed specifically for Real-Time Strategy (RTS) Games, where we conducted most of our experiments. After identifying some shortcomings of this first agent, a second agent was developed to help us better investigate the issues. This time, the agent was tailored for Frozen Lake – detailed in Section 4.2.3 –, a simpler, single-agent RL environment. This second agent not only served as a tool to verify and evaluate the efficacy and potential of our proposed architecture, but also allowed us to understand better the limitations observed in the first agent.

4.2.1 Real-Time Strategy Model

In recent years, a variety of reinforcement learning algorithms have emerged to address complex decision-making tasks. Among these, Proximal Policy Optimization (PPO) stands out as a particularly effective approach for developing autonomous agents in challenging environments. Its effectiveness has led to widespread adoption in both research and practical applications. Our proposed approach leverages PPO due to its key advantages over alternative methods, including stability, ease of implementation, and adaptability across diverse domains.

PPO's design prioritizes stability throughout training, employing a surrogate objective function with an integrated clipping mechanism. This approach prevents excessive policy updates, which could lead to training instability. By constraining the policy updates, PPO reduces the likelihood of divergence or oscillation, leading to a more stable learning process. Furthermore, PPO is adaptable to various types of reinforcement learning tasks, including continuous and discrete action spaces. This flexibility allows for the development of a framework capable of addressing a diverse range of applications, making it an attractive choice for researchers and developers seeking a broadly applicable solution. PPO has also demonstrated impressive performance in various complex game environments, including Starcraft II [Vinyals et al., 2019] and Dota 2 [Berner et al., 2019].

These characteristics collectively contribute to why PPO emerged as the preferred

algorithm for developing robust and competitive AI agents in this domain, which lead to the development of some impressive autonomous agents in μRTS as we have seen in the previous chapter.

Given our focus on RTS games, it is crucial to employ a neural network architecture capable of managing multiple units simultaneously. The GridNet architecture, which we discussed on Section 4.1.1, is an excellent choice for our purposes. It allows for the assignment of individual actions to each cell within a grid-based framework, aligning seamlessly with the structure of our main framework: μRTS.

As shown in Figure 4.3, our model architecture is divided into three parts: one encoder and two decoders. The encoder is composed of convolutional layers followed by traditional max pooling layers; It takes the agent's observation as input and generates concise feature maps that can be better used to predict the policy the agent should follow in the given environment state, as well as the state's value. While the original Gridnet model encoder encompasses four pairs of convolutional and pooling layers, we noticed that downsizing it to two pairs of layers maintained the agent's performance but significantly reduced the number of parameters on the network. The resulting encoder consists of the following layers:

1. **Convolution:** 32 filters, kernel size 3×3, padding 1, stride 1;
2. **Max Pooling:** window size 3×3, padding 1, stride 2, ReLU activation;
3. **Convolution:** 64 filters, kernel size 3×3, padding 1, stride 1;
4. **Max Pooling:** window size 3×3, padding 1, stride 2, ReLU activation.

All layers in this architecture (encompassing one encoder and two decoders) are initialized by applying orthogonal weight initialization [Saxe et al., 2013] to their weights, using a standard deviation of $\sqrt{2}$, and setting their biases to zero.

The encoder output is used by the first decoder – the actor's decoder – to generate the probabilities of each action for each cell of the grid. This is done by combining transposed convolutional[2] layers with pooling layers, resulting in an output tensor with the same height and width as the initial observation but with a different number of channels. Each output channel is responsible for a parameter of the possible actions and will be used to compose the action the agent will perform. This first path of the network behaves in the same way as the grid-wise control described previously, taking a grid observation as input and outputting an action for each grid cell. Once again, we reduced the reference GridNet actor by removing the last two transposed convolutional and pooling layers. The actor's decoder is composed of the following layers:

[2] Also called deconvolutional layers.

1. **Transposed Convolution:** 32 filters, kernel size 3×3, padding 1, output padding 1, stride 2, ReLU activation;
2. **Transposed Convolution:** 78 filters, kernel size 3×3, padding 1, output padding 1, stride 2.

The second decoder – the critic's decoder – uses the encoder output to predict the state's value function. The reference architecture is composed only of fully connected layers, and it flattens the encoder output to pass it onto the critic's decoder. Instead of flattening the encoder's output, we added an SPP layer before the fully connected layers. This new layer generates a standardized representation of any input passed to the critic, which allows the agent to work properly in any environment regardless of the observation dimensions. Despite being a simple innovation, this causes great changes to the actor's behavior and, as shown in our experimental results, improves the agent's effectiveness in most scenarios. The critic's network is structured as follows:

1. **Spatial Pyramid Pooling:** single sub-layer with 4×4 bins (for experiments on different configurations for this layer, see Chapter 5);
2. **Fully Connected:** 128 units, ReLU activation;
3. **Fully Connected:** 1 unit.

Overall, the revised architecture with an optimized encoder and actor's decoder, as well as an enhanced critic's decoder via the SPP layer, not only reduces the number of parameters in the network but also boosts the model's scalability and adaptability in various environment states, demonstrating robust improvements in the agent's performance as we will see on our evaluation chapter.

4.2.2 Framework for Diverse Training Environments

The new model's capabilities allow a distinct training approach. We can leverage the agent's expertise in one scenario to accelerate its learning process in other similar settings. We take this idea one step further and utilize multiple scenarios in a single training set to develop a more general agent that learns to perform well in distinct environments.

We introduce a framework that uses dynamic environment selection before each training iteration to optimize the agent's exposure to different scenarios. This framework can be used with many algorithms, such as PPO or DQN. The chosen algorithm collects experience by interacting with the environment and optimizes its parameters as

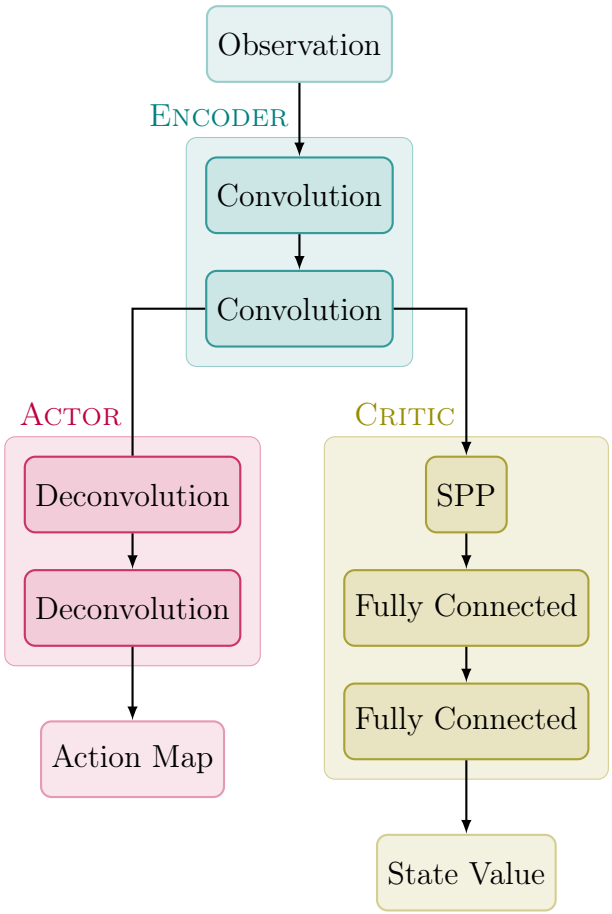

Figure 4.3: Proposed network architecture for the μRTS Environment. The encoder processes observations of the environment and passes the encoded data to both the actor and critic components. The actor generates action probabilities, while the critic estimates the value function of the current state. Inspired by the Gridnet architecture, this design incorporates a SPP layer in the critic component, enabling the network to handle different map sizes.

Algorithm 1 Environment Swap Framework

1: **for** iteration $= 1, 2, \ldots$ **do**
2: Select environment E according to strategy Q
3: Collect experience in E for T timesteps by following algorithm ALG_θ
4: Update θ using collected experience according to ALG_θ procedures
5: **end for**

usual, while the framework is responsible for selecting and providing an environment every iteration. Algorithm 1 shows a pseudocode for a generic version of this framework.

As we will see in Chapter 5, the strategy used to select the environment may also affect the agent's behavior. Different approaches may lead to more consistent learning throughout all environments or prioritize specific environments without forsaking others. For example, let us consider a set of three maps $\{A, B, C\}$, with dimensions $(a \times a)$, $(b \times b)$, and $(c \times c)$, respectively. We could sequentially cycle through $\{A, B, C\}$, ensuring

Algorithm 2 PPO with environment swap

for iteration $= 1, 2, \ldots$ **do**
  Select environment E according to strategy Q
  for actor $= 1, 2, \ldots$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment E for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size M
  $\theta_{old} \leftarrow \theta$
end for

the agent would experience all three distinct representations for the same amount of steps. Another option would be to randomly select the maps following a uniform distribution, creating an unpredictable experience for the player.

Moreover, we can assign different values / weights to the performance in different scenarios. For example, if we evaluate the agent by his weighted average performance, with weights $(0.6, 0.2, 0.2)$ for (A, B, C), we would prioritize experience on map A. We could tailor the selection strategy to meet this specific need by utilizing a weighted random selection with the same $(0.6, 0.2, 0.2)$ weights, ensuring map A would be prioritized. The strategy selection could also be used as a form of Curriculum Learning [Bengio et al., 2009], where the agent starts playing only in simple maps and, as time passes, more complex maps are added to the selection pool.

In this work, we apply this framework to the the PPO algorithm specifically. The modification to the algorithm can be seen in Algorithm 2

4.2.3 Frozen Lake Model

Our second model was tailored to an agent tasked with navigating a grid-based environment. Within this setting, we made the assumption that the agent possessed complete visibility of the grid, encompassing its dimensions, obstacles, goals, and its own position. In an environment characterized by these features and fixed-size inputs, a conventional Q-Learning could be employed to address the navigation challenge. However, the inherent limitation arises when dealing with varying input sizes. Traditional Q-Learning relies on a table of Q-values tied to specific state-action pairs, thereby becoming constrained by input dimensions. Alternatively, a DRL approach could be employed, with DQN emerging as the most straightforward option rooted in Q-Learning principles. However, it would also encounter difficulties with diverse input sizes, as the convolutional layers in DQN would generate multiple output sizes, which is incompatible with the ex-

pected dimensions of the fully connected layers responsible for outputting the Q-values.

In the progression of our research, the initial application of the PPO algorithm within a RTS game environment served as a robust framework for exploring complex decision-making processes. However, upon encountering specific learning challenges within this context (detailed on the next chapter), our investigation shifted towards a more granular analysis using the Frozen Lake environment.

The Frozen Lake environment presents a seemingly simple challenge: an agent, represented as a character on a grid map, must navigate from a starting point to a goal without falling into holes scattered throughout the lake. Despite its simplicity, the Frozen Lake present several key features that make it a valuable tool for RL research, as discussed in Section 2.5. Moreover, due to its grid-based maps, it can also be easily integrated with our scale-invariant model. By employing maps of varying scales, we can systematically escalate the complexity of the navigation task. Larger grids demand the agent to acquire proficiency in learning extended sequences of actions and managing larger state spaces, all while retaining its ability to solve smaller-scale maps. Furthermore, drawing parallels between the frozen lake navigation task and the micromanagement of units in a RTS setting becomes evident. In an RTS, akin to navigating the Frozen Lake, players must control units to maneuver through obstacles and reach their objectives.

For the Frozen Lake environment, we decided to adopt the DQN algorithm. This decision was based on the fact that our approach is not attached to a single algorithm and that the Frozen Lake environment is typically solvable by employing a traditional Q-Learning agent. Using the DQN allowed us to demonstrate the versatility of our approach while staying closely aligned with well-established solutions for this environment. The DQN's value-based approach also streamlines the agent's architecture, facilitating a clearer understanding of how our modifications influence its behavior.

In this new environment, we also implemented an adaptation of the conventional CNN architecture, integrating a Spatial Pyramid Pooling (SPP) layer situated between the final convolutional layer and the network's first fully connected layer. This design mirrors the approach adopted for both the encoder and critic components of the μRTS model. The SPP layer is designed to process inputs of varying dimensions, outputting a uniform-sized representation irrespective of the input size. This combination of convolutional layers and SPP layers yields a versatile and powerful encoder, capable of producing standardized representations across diverse environments that we can then use to infer the Q-values of the state-action pairs (s, a). It is important to highlight that this approach does not employ any form of input preprocessing, thereby preserving the entirety of the state's information without introducing any distortions.

In the classic Frozen Lake challenge, the agent's objective is to learn how to navigate across a singular map, traditionally encoding the state as a single integer that represents the agent's location. However, to accommodate for deep learning approaches, we

revised the state representation strategy. Instead of relying on a singular integer, our approach utilizes three binary feature planes for the observation model. Each plane adheres to dimensions $(h \times w)$ of the map, where h and w are the height and width, respectively. The first plane marks the agent's current location; the second delineates the positions of the holes; while the third highlights the goal's location. This adjustment ensures our agent is equipped to intelligently navigate any map on the Frozen Lake environment.

Given the inherent simplicity of the Frozen Lake environment, which is typically solvable through tabular methods, we tried to avoid overcomplicating our model. Consequently, in our preliminary approach, we adopted a basic CNN design for the DQN. This design comprises a single convolutional layer, succeeded by a traditional pooling layer. These layers are then connected to a hidden fully connected layer, culminating in a final output layer responsible for generating the Q-values associated with each possible state-action pair.

In designing this preliminary network, we intentionally configured the initial layers to act as an encoder, simplifying the detailed observation of the map into an abstracted representation. With this goal in mind, we initially matched the dimensionality of the hidden fully connected layer to the total count of possible states within the map. For instance, in a setting with an 8×8 map, there are 64 unique states, prompting us to set the hidden layer's size to 64 units. This configuration allows our agent to undertake dual responsibilities: first, to learn the encoding of observations into their respective states efficiently, and second, to learn the most promising action for each state based on the expected outcomes. This preliminary version of the model was developed to validate that the adapted representation for the Frozen Lake environment was adequate and that deep learning algorithms could learn how to reach the goal with it.

Upon verifying that our preliminary model demonstrated the capacity to learn effectively on smaller maps with the initial architecture, we introduced a Spatial Pyramid Pooling (SPP) layer situated between the pooling and the fully connected layers. The primary role of the SPP layer is to unify the output from the pooling layer across varying map sizes, thereby equipping our agent with the ability to apply its learned insights universally, regardless of the map dimensions.

To further enhance the model's learning ability, we incorporated reward shaping tailored to the specifics of the Frozen Lake environment. The original reward function of the Frozen Lake only rewards the agent when he reaches the goal. Our new reward function was designed to provide more granular feedback to the agent, rewarding not only the successful navigation to the goal but also intermediate milestones such as avoiding holes and progressing towards the goal. Specifically, we introduced small positive rewards for each step that brought the agent closer to the goal and small penalties for steps that either moved the agent away from the goal or brought it dangerously close to holes. Additionally, we incorporated a negative reward for falling into a hole and a significant

positive reward for successfully reaching the goal. By providing continuous and context-sensitive feedback, we ensured that the agent's learning process was not only focused on end goals but also on optimizing the path taken to achieve these goals, thereby enhancing the agent's overall performance and resilience in varying and complex scenarios.

To isolate the learning challenges faced by the RTS model, we opted against incorporating our framework for diverse training environments to the DQN algorithm of this second agent. This approach allows us to gain a clearer understanding of the model's internal workings and pinpoint the specific issues hindering its performance.

Chapter 5

Evaluation

As detailed in the previous chapter, our focus on scale-invariant reinforcement learning led to the creation of two distinct approaches, each tailored to address specific challenges. This chapter is explains the evaluation process undertaken for each agent and presents the findings derived from our experiments. Furthermore, we provide insightful interpretations of the results, explaining their meaning and importance.

5.1 μRTS

We verified our model's efficacy in several experiments on Gym-μRTS, validating its performance against established agents. We followed the setup employed by Huang et al. [2021] and trained our model against CoacAI, RandomBiasedAI, LightRushAI, and WorkerRushAI. Using these diverse opponents grants a more complete experience for the agent and prevents it from losing to some simple but effective strategies. We also designated CoacAI[1] as the main opponent in our experiments. CoacAI is the winner of the 2020 CoG MicroRTS Competition, and the other bots used for training are part of the μRTS framework and are used as baselines for the competitions. While the other opponents are not as relevant as CoacAI, playing against them ensures our agent will acquire knowledge of many strategies and will not easily lose to different opponents.

Furthermore, we use the best model[2] developed by Huang et al. [2021] as a baseline to compare and evaluate our approach and the impact of the proposed modifications. Unless stated otherwise, our proposed model used SPP with a single-sublayer with 4x4 bins and trained by following Algorithm 2 with a sequential map selection where maps were swapped every 100,000 steps. As we show below, this was the best configuration we have found.

[1] Available at https://github.com/Coac/coac-ai-microrts

[2] The model did not receive an official name but was referred to as the combination of GridNet + PPO + invalid action masking + diverse bots + encoder-decoder.

All agents were trained for 300 million steps, with the resulting policy being tested in 100 games against CoacAI. Three map configurations were used, with sizes 8×8, 16×16, and 24 × 24. Since the baseline model cannot play in multiple map dimensions without changes to its architecture, we have used three versions, one for each map, adapting the first fully connected layer of their critic to accommodate the encoder's output. Each agent configuration used in the experiments was trained only once due to infrastructure and time constraints. Although a single training is not ideal for the evaluation of reinforcement learning agents – given the high variance typically associated with this method – the results we obtained can be interpreted as preliminary insights into the agents' performance. These findings provide a useful starting point for further investigation and highlight the need for additional training runs to assess the robustness and consistency of the agents across varying scenarios.

We also conducted hyperparameters searches to identify the optimal hyperparameters for our model. However, we found that the hyperparameters used by Huang et al. [2021] also yielded the best performance for our model. Therefore, we decided not to include these preliminary searches in our findings. The hyperparameters employed in this evaluation are presented in Table 5.1. All the implementation details used on the experiments can be found at our GitHub repository[3].

Hyperparameter	**Value**
Learning rate	2.5×10^{-4}
Learning rate schedule	Linear annealing
Discount factor	0.99
Advantage normalization	Yes
Entropy coefficient	0.01
Value function coefficient	0.5
PPO surrogate clipping	0.1
Policy update epochs	4
Batch size	6144
Minibatch size	1536

Table 5.1: Hyperparameters for model evaluation.

The game results during test time were the primary metrics used to evaluate the agents. For easier visualization, we opted to simplify it, and instead of using the raw results, we employ a Score metric where a player gets 1 point for each victory and 0.5 points for each draw.

In RL, agents receive rewards for completing certain actions or reaching specific states, generally associated with their final goal. Since we employ reward shaping, our agent receives rewards from several small actions, such as collecting resources or attacking

[3]Available at https://github.com/marcelo-lemos/MicroRTS-Py

enemy units. One of the main reward components we use is a win/loss reward, which is always received at the end of a game: 1 in case of a win, 0 in case of a draw, or -1 in case of a loss. The sum of all rewards received across a game (or episode) is called the episodic return, which does not have an upper bound in our case but has a lower bound of -1 in case the only reward received was of a loss. We use the win/loss rewards and episodic returns received during training as additional metrics to analyze learning progression. Since our agent plays thousands of games during training, we apply a moving average to better visualize our graphs. For the win/loss rewards, values closer to -1 indicate the agent is losing more games than winning, while values closer to 1 indicate the opposite. Values close to zero indicate the agent is winning almost as much as it is losing.

Our experiments are divided into four categories as follows.

5.1.1 Proposed vs Baseline Model

In this first experiment, we compare our proposed model with the best version developed by Huang et al. [2021] playing against CoacAI in two different scenarios: specialized and generalist described next.

5.1.1.1 Specialized Scenario

For each of the three maps, an agent of each model was trained and tested exclusively on it. As shown in Figure 5.1, the proposed model outperformed the baseline in all three maps. The greatest difference occurred on the 8×8 map, where the original lost all 100 games against CoacAI, while the proposed version achieved a score of 70 points. This disparity highlights a critical limitation of the baseline model: its architecture and hyperparameters were tailored for the 16×16 map, which likely caused suboptimal performance in environments requiring different strategies. The 16x16 and 24x24 maps are relatively large, causing the games to last longer and requiring strategies that focus on developing more military structures and combat units. On these maps, long-term planning and gradual build-up are essential for success. In contrast, the 8x8 map is significantly smaller, favoring quicker strategies. Here, efficiency in creating a maximum number of simple workers and deploying them for immediate attacks becomes crucial. The baseline model, designed for the larger 16x16 environment, struggles with the rapid, aggressive tactics

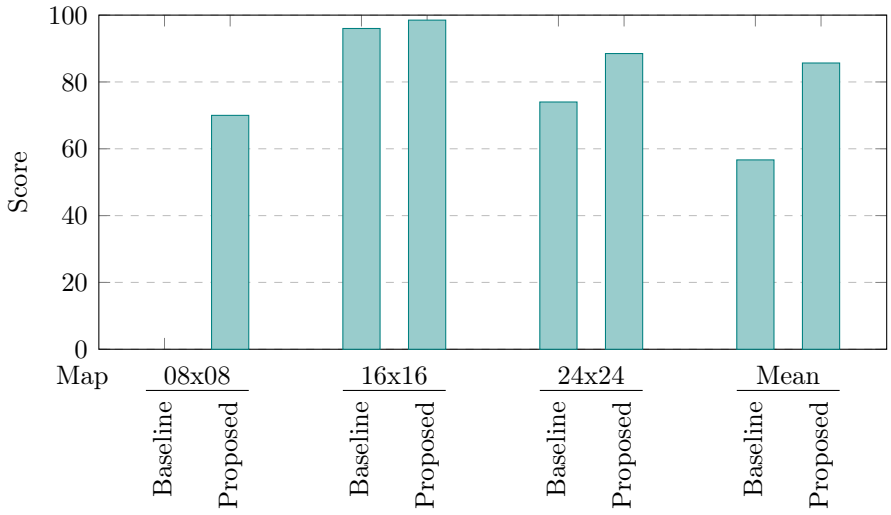

Figure 5.1: Comparison of proposed and baseline models' scores across three map dimensions in the specialized setting. The proposed model outperforms the baseline on all tested dimensions.

needed for the 8x8 map, whereas the proposed model adapts better to these distinct scale demands.

Figures 5.2 and 5.3 show that the win/loss reward and the episodic return are very close for the two models tested, except for the 8×8 map once again. On this smaller map, the baseline model failed to learn effective gameplay strategies, as evidenced by its declining performance after 100 million steps, from which it could not recover. This suggests a critical deficiency in the baseline model's adaptability and robustness when confronted with different environmental characteristics.

Our novel architecture, designed with greater flexibility, demonstrates consistent performance across all tested environments. It does not compromise effectiveness when focusing on a single environment and shows improved results. We also notice that the episodic returns differ from one map to another, which can be attributed to our use of reward shaping. In larger maps, units must traverse greater distances, leading to longer games and allowing the agent to accumulate more rewards from a series of smaller actions. This accumulation contrasts with the more immediate, direct rewards available in the shorter games of smaller maps.

5.1.1.2 Generalist Scenario

The proposed model can train across multiple maps in a single training session, whereas the baseline model can only train on one map at a time. For this experiment, we

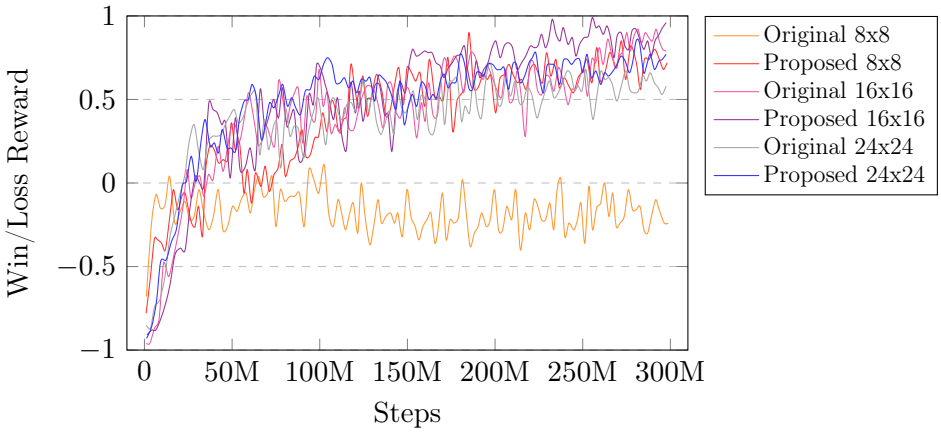

Figure 5.2: Comparison of the moving average of the win/loss rewards between proposed and baseline models across three map dimensions in the specialized setting. Both models exhibit similar performance trends, except on the 8×8 map, where the baseline model struggled with learning optimal winning strategies over time.

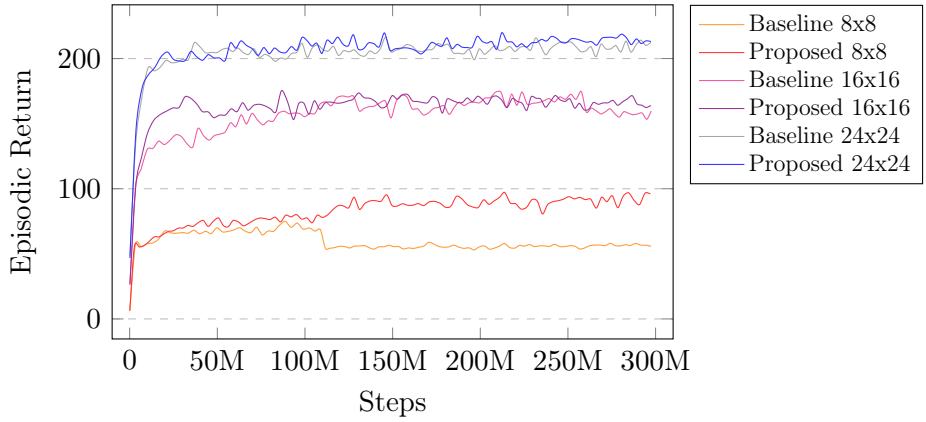

Figure 5.3: Comparison of the moving average of the episodic returns between proposed and baseline models across three map dimensions in the specialized setting. Both models exhibit similar performance trends, except on the 8×8 map, where the baseline model struggled with learning optimal winning strategies over time. Larger maps result in longer games, leading to greater cumulative rewards per game for the agents.

compare the results of one instance of the proposed model, which was trained across three maps, against three separate instances of the baseline model, each trained on a different single map.

Figure 5.4 shows that the specialized agents of the baseline model outperformed our generalist agent in two of the three maps. On 16×16, the baseline received 36.5 more points than the proposed model. However, on the 24×24, the performance gap was much narrower, with the baseline only scoring 3.5 points higher. Lastly, on the 8×8 map, the proposed model achieved 97 points against zero of the baseline. When considering all maps, our proposed model achieved a better mean score, almost 20 points higher than the baseline.

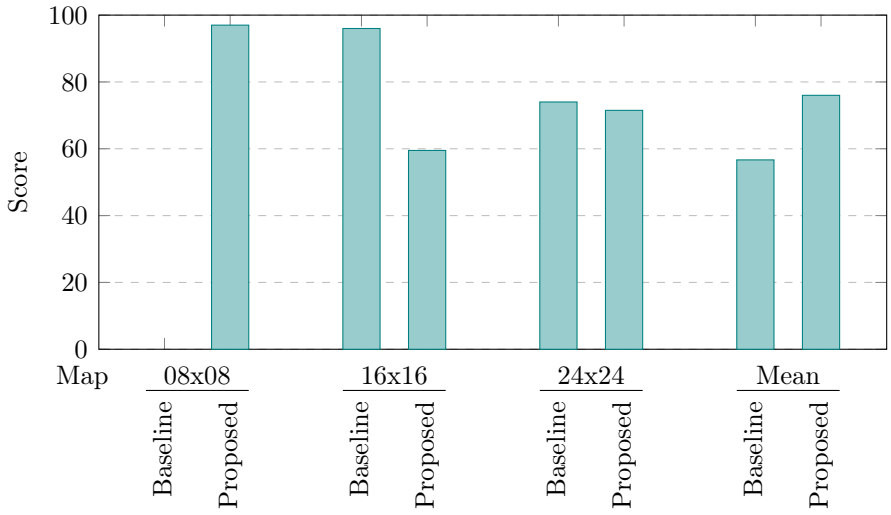

Figure 5.4: Comparison of proposed and baseline models' scores across three map dimensions in the generalist setting. The baseline agent outperforms the proposed model in two of the three maps, but the proposed model achieves a greater mean score.

Note that the proposed model had a third of the total training budget of the baseline model in this setting. The baseline had to be trained in each map individually for 300 million steps, totalling 900 million. In contrast, the proposed model trained a single time for 300 million total. Despite this significantly reduced training budget, the proposed model demonstrated a notable performance improvement on average.

Figure 5.5 shows that the win/loss reward received by the generalist agent is very similar to the baseline on maps 16×16 and 24×24. This suggests that our proposed generalist model is nearly as effective as the baseline when considering the 24×24 map. Figure 5.6, however, shows that the generalist agent's episodic returns are biased towards the 8×8 baseline. Since the generalist agent is trained on all three maps for the same number of steps, and because the 8×8 map is smaller and games are shorter, the agent completes more episodes on this map, skewing the curve towards the 8×8 baseline.

In conclusion, while the baseline model's specialized agents perform slightly better on larger maps, the proposed generalist approach offers superior overall performance and efficiency across diverse environments. The ability to generalize across different map sizes and strategies demonstrates a significant advantage in scenarios where agents must perform well in diverse settings. However, the training routine must be carefully crafted to prevent biases towards some maps or strategies.

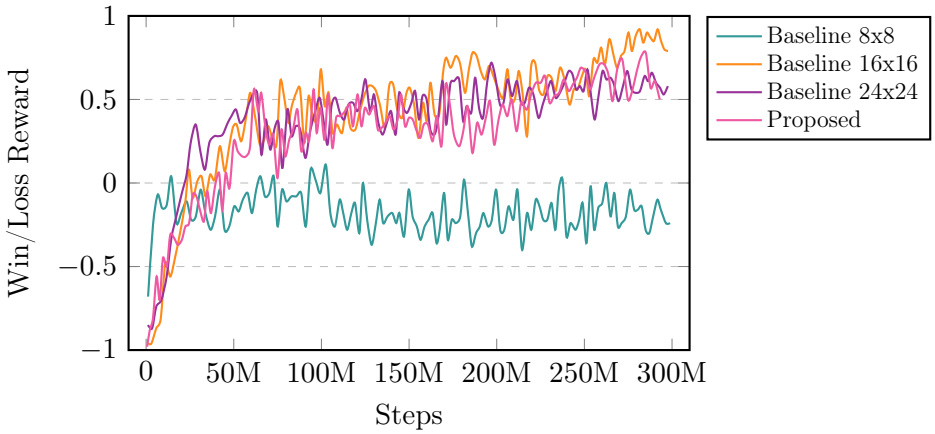

Figure 5.5: Comparison of the moving average of the win/loss rewards between proposed and baseline models across three map dimensions in the generalist setting. Both models exhibit similar performance trends, except on the 8×8 map, where the baseline model struggled with learning optimal winning strategies over time.

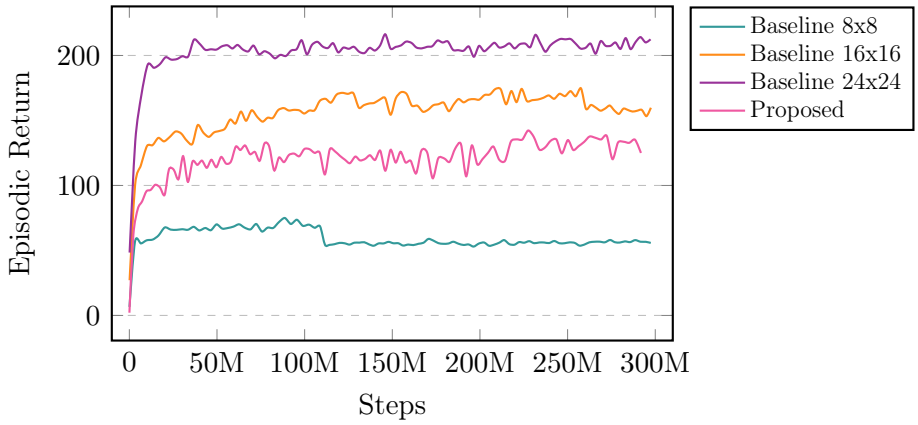

Figure 5.6: Comparison of the moving average of the episodic returns between proposed and baseline models across three map dimensions in the specialized setting. The proposed model's performance cannot be directly compared to the baselines' due to its training on multiple map dimensions, each offering distinct maximum achievable cumulative rewards.

5.1.2 Specialized vs General Training

To consolidate whether training in various scenarios instead of focusing on a single environment is advantageous for the agent, we tested four agents of our proposed model with the same architecture and configuration, each trained in different map settings: (i) 8×8 map only, (ii) 16×16 map only, (iii) 24×24 map only, and (iv) all three maps. For clarity, from now onward, we will refer to each agent as S-08x08, S-16x16, and S-24x24 for those trained solely on maps 8×8, 16×16, and 24×24, respectively. The agent trained on all maps will be referred to as *generalist*. Despite the different training, they were all

evaluated on all three maps and we also included a 32×32 map in the evaluation that was not included in any agent's training.

Figure 5.7 shows that the specialized agents outperformed the generalist agent in two of the three maps. On 16×16, the S-16x16 received 39 more points than the generalist, but on the 24×24, the difference was a lot smaller, with the S-24x24 being only 17 points better. However, on the 8×8 map, the generalist outperformed the S-08x08, achieving 97 points compared to the 70 points of the specialized agent. Lastly, on the 32×32 map – which was not seen during training – the generalist achieved the second-best performance, with 53 points against the 65 points of the S-24x24. All four maps present the same structure and units, the only difference being the map's scale. This factor gave an edge to S-24x24 due to the similarity in the scale of the training map and the 32×32 map. Considering all maps, the generalist agent achieved a better mean score, 17 points higher than the second place.

Figures 5.8 and 5.9, illustrate the win/loss reward and episodic return, respectively, where we note patterns similar to those observed in the previous experiment. A major difference was the performance of the S-08x08 agent, which, unlike the baseline model, managed to attain good results on map 8×8. The flexibility of our architecture allowed the S-08x08 to adapt better to the specific demands of the smaller map.

While specialized agents may have an edge on their respective training maps, the generalist agent demonstrates an overall superior performance. The generalist's ability to adapt and perform well across different environments, including the previously unseen 32×32 map, highlights the advantages of training on multiple scenarios. This adaptability suggests that a more diverse training approach can lead to more robust performance in varied settings, compared to agents trained exclusively on single environments. This approach ensures adaptability and robustness but also prepares the agent for unforeseen challenges, making it a more versatile and efficient solution for diverse environments.

5.1.3 Environment Selection

As discussed before, our training method involves swapping environments mid-training. To investigate the impacts of the strategy used to select the new environment, we examined both the method and the frequency of environment selection.

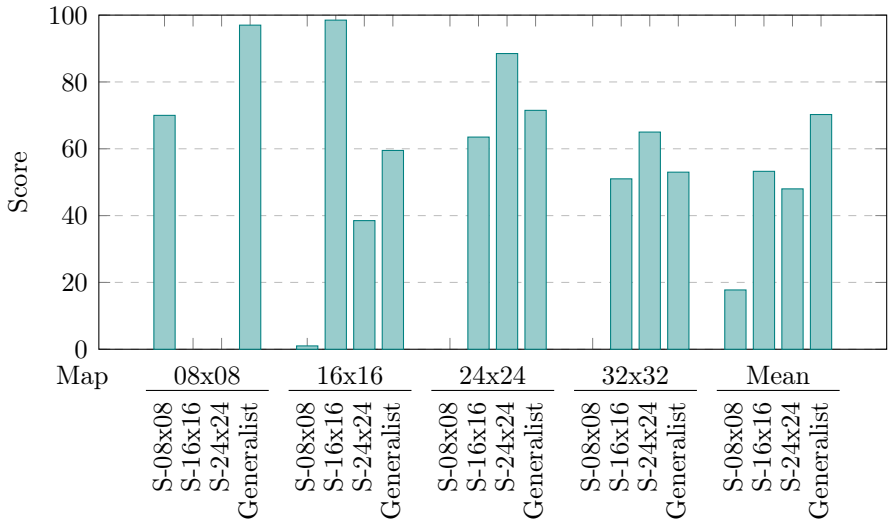

Figure 5.7: Comparison of specialist and generalist models' scores across four map dimensions. The generalist model exhibits the best or second-best performance on all maps and achieves the best performance overall.

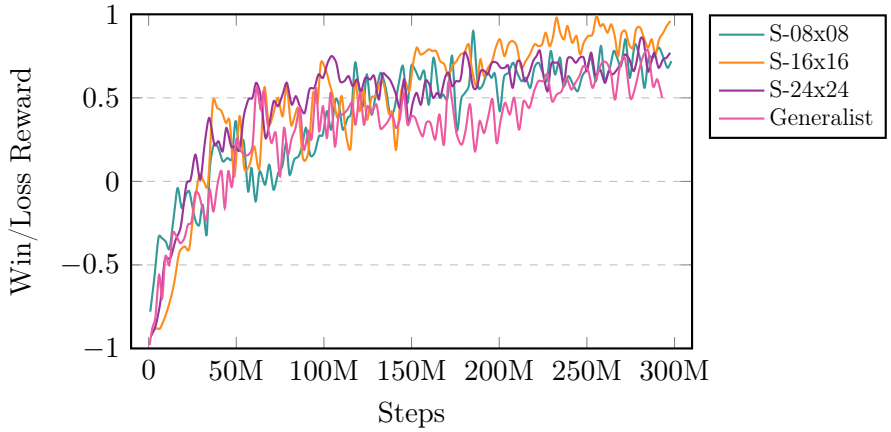

Figure 5.8: Comparison of the moving average of the win/loss rewards between specialized and generalist models across three map dimensions during training. All models exhibit similar performance trends.

5.1.3.1 Selection Method

Two methods were verified, random and sequential. In the sequential method, we cycle through a predefined sequence of maps from smallest to largest.

As seen in Figure 5.10, the sequential selection method outperformed the random method in all three maps tested. This difference was especially pronounced on the 8x8 map, where the random method lost all 100 games, whereas the sequential method won 97 games. The episodic return of both methods, depicted in Figure 5.11, reveals a significant learning instability for the random method compared to the sequential method. This

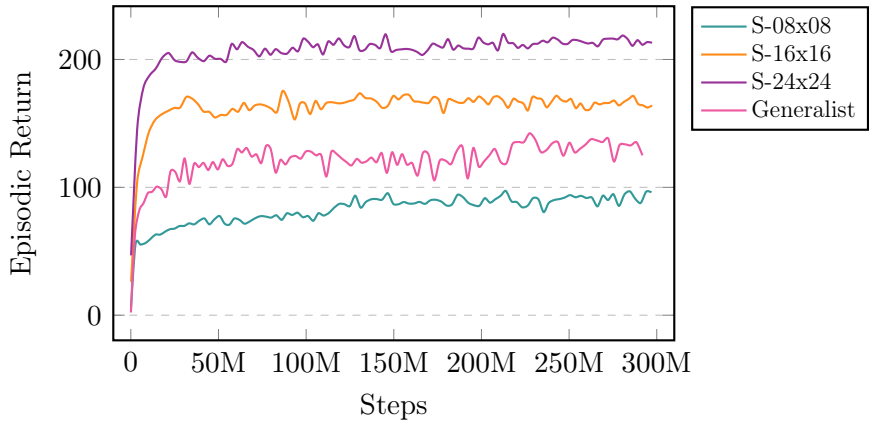

Figure 5.9: Comparison of the moving average of the episodic returns between specialized and generalist models across three map dimensions during training. The episodic returns of the generalist model fall between those of the S-08x08 and S-16x16 models, reflecting intermediate cumulative rewards due to its training on multiple map dimensions.

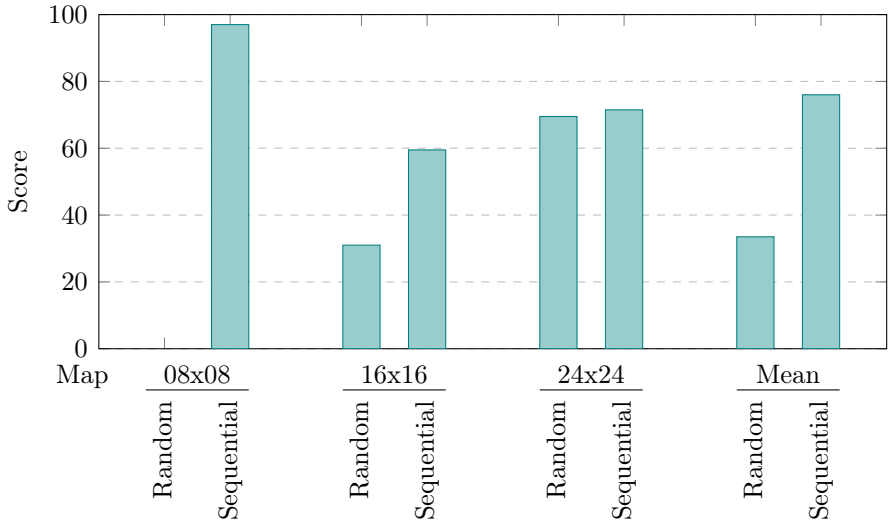

Figure 5.10: Comparison of models' scores using random and sequential environment swap across three map dimensions. The sequential method consistently outperforms the random method in all maps.

instability directly correlates to the reduction of the win/loss reward in Figure 5.12, ultimately affecting the resulting policy. Notably, sequential selection promotes a more consistent and smoother learning process.

The sequential method's superiority can be attributed to its structured progression, which likely allows the agent to develop and reinforce strategies incrementally. By starting on smaller maps and gradually moving to larger ones, the agent can build on its previous experiences, leading to more stable and effective learning. In contrast, the random method's lack of structure can cause the agent to randomly encounter vastly different environments, disrupting the learning process and leading to instability.

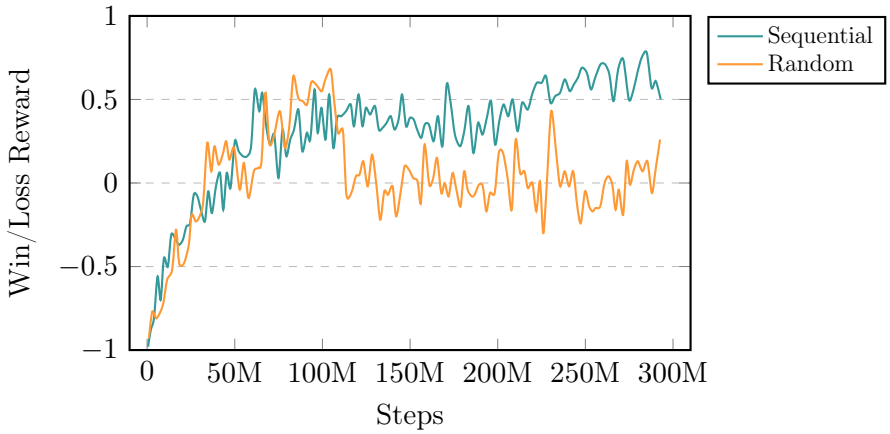

Figure 5.11: Comparison of the moving average of the win/loss rewards between models using random and sequential environment swap. The sequential method demonstrates a consistent improvement trend over time, whereas the random method experiences significant performance drops.

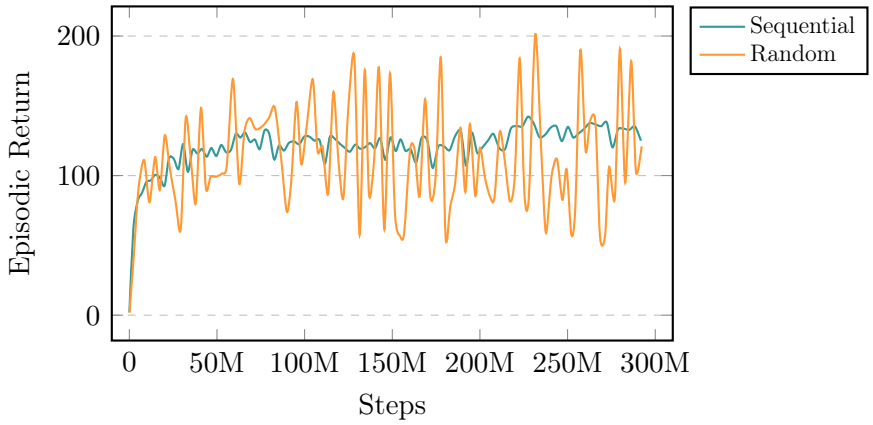

Figure 5.12: Comparison of the moving average of the episodic returns between models using random and sequential environment swap. The random method exhibits unstable returns compared to the sequential method.

5.1.3.2 Change Frequency

To evaluate the change frequency's impact, we utilized only the sequential method and ensured our agent experienced each environment for the same total steps. We tested two different frequencies, one changing every 100 million steps – causing each environment to be seen a single time – and one changing every 100,000 steps. We refer to them as A-100M and B-100K, respectively.

As seen in Figure 5.13, the agent B-100K, trained with more frequent swaps, achieved better results on the test games on most maps, except for the 24×24 map, where the score was 3.5 points below A-100M. Figure 5.14 shows that the agent that trained

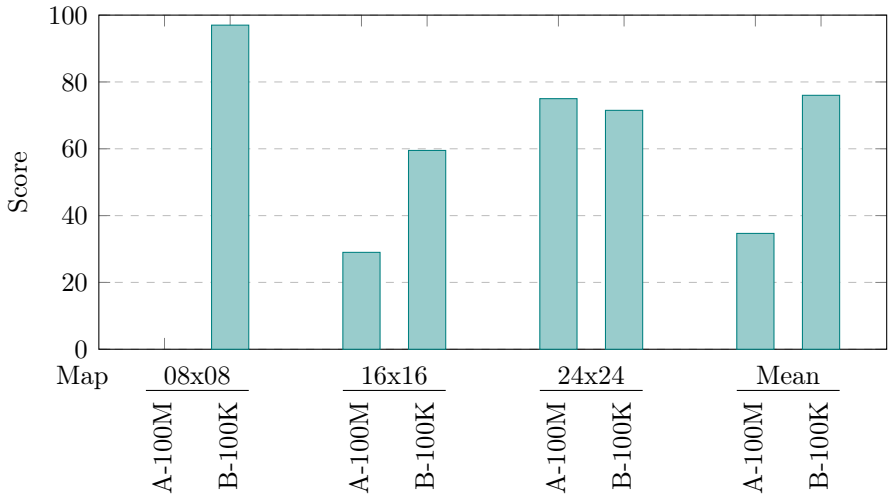

Figure 5.13: Comparison of models' scores when the environment is swapped every 100K or 100M steps across three map dimensions. Swapping every 100K steps generally results in better performance, except on the 24×24 map, where swapping every 100M steps yields superior results by a small margin.

for longer periods before swapping environments presented a big drop in performance during the change of context. This suggests that the agent specialized in a single map after training on it for an extended period, but struggled to adapt when the environment changed, taking time to adjust its strategies to the new situations.

Figure 5.15 shows the episodic return, where we can clearly see when the map changes occurred for the A-100M agent, marked by noticeable increases on the episodic return. In contrast, the more frequent map changes for agent B-100K led to smoother and more consistent learning. This continuous exposure to different environments helped the agent maintain flexibility and prevented overfitting to any single map.

The agent trained with more frequent swaps (B-100K) achieved better overall results and demonstrated smoother and more consistent learning. This suggests that maintaining a dynamic and varied training regimen, with frequent exposure to different environments, is crucial for developing agents capable of adapting to multiple scenarios encompassing different scales.

5.1.4 SPP Layer Size

We study the impact of different sizes of the SPP layer in our agent by comparing four different compositions. The sub-layers tested are (i) a single sub-layer with 2×2 bins, (ii) a single sub-layer with 4×4 bins, (iii) a sub-single layer with 8×8 bins, and

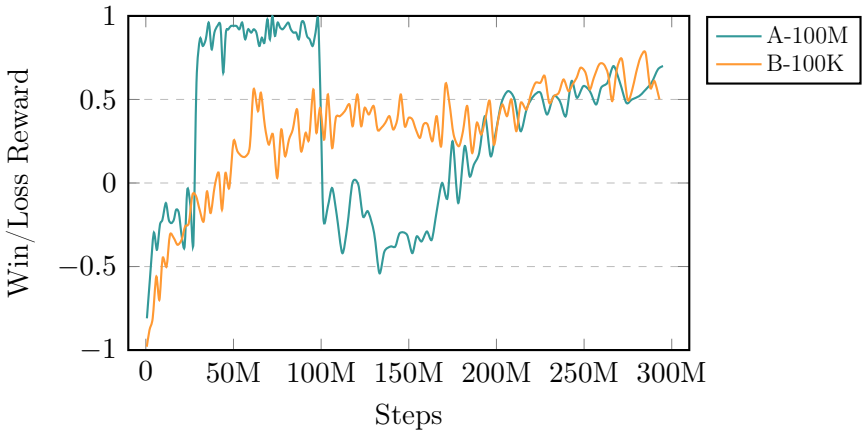

Figure 5.14: Comparison of the moving average of the win/loss rewards when the environment is swapped every 100K or 100M steps. The model A-100M stays for longer periods on the same map and exhibits significant performance drops when the environment is swapped. Meanwhile the model B-100K exhibits more consistent improvements.

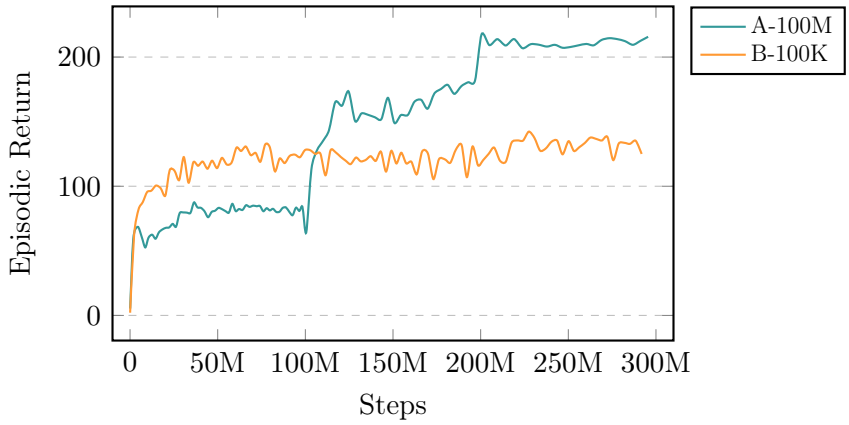

Figure 5.15: Comparison of the moving average of the episodic returns when the environment is swapped every 100K or 100M steps. Model B-100K, which experiences more frequent swaps, displays a smoother curve. In contrast, Model A-100M shows distinct steps in episodic returns corresponding to each environment swap.

(iv) three sub-layers with 2×2, 4×4, and 8×8 bins. All four architectures were trained and tested on all three map dimensions.

As shown in Figure 5.16, single sub-layer architectures with 2×2 and 4×4 bins attained the best results, with mean scores of 73.5 and 76, respectively. In contrast, the bigger architectures exhibit worse performances, especially on map 16×16. The small single sub-layers displayed better generalization than bigger or multiple ones. Figures 5.17 and 5.18 show that all four configurations performed closely during training. The biggest single sub-layer, with 8×8 bins, deviated more from the others. Its win/loss reward received dipped around 80M and 260M steps. It eventually recovered from the first dip but not the second one, which surely impacted the final policy.

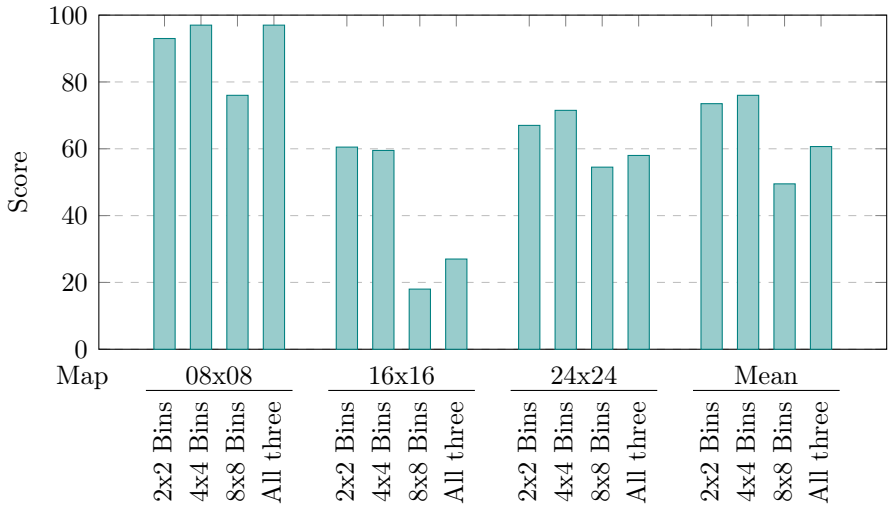

Figure 5.16: Comparison of models' scores using different SPP sub-layers across three map dimensions. The model with 4×4 bins exhibits the best performance overall and is closely followed by the model with 2×2 bins. The model with 8×8 shows the lowest performance among all configurations.

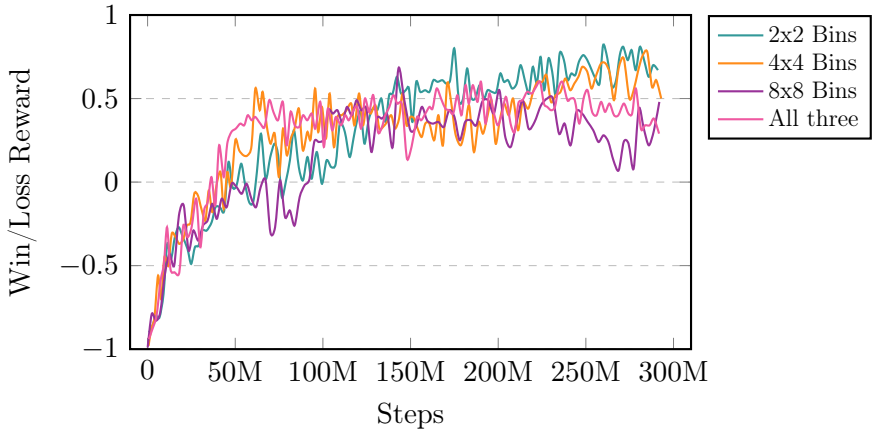

Figure 5.17: Comparison of the moving average of the win/loss rewards between models using different SPP sub-layers. All models exhibit similar performance trends.

5.2 Frozen Lake

In the Section 5.1, we observed a learning instability on smaller maps, notably on the 8×8 size. The tested agents exhibited either exceptionally high or exceedingly low score ratings on the smallest map. Building upon this observation, we devised experiments for the Frozen Lake environment (described on Section 2.5) that would allow us to investigate this issue in more detail. Notably, these experiments utilized our second model, which employs Deep Q-Networks (DQN) rather than PPO. This section presents and discusses the experiments conducted to evaluate the effectiveness and scalability of

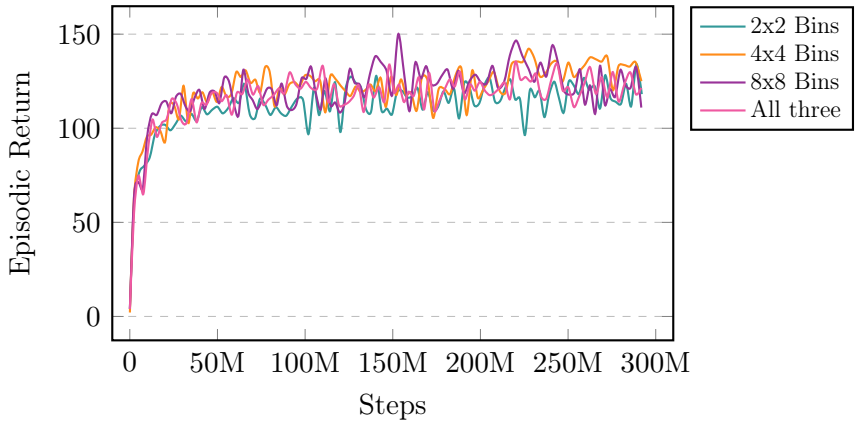

Figure 5.18: Comparison of the moving average of the episodic returns between models using different SPP sub-layers. All models exhibit similar performance trends.

our proposed architecture in the Frozen Lake environment.

The primary focus of these experiments was to examine the model's capability to generalize its learning from smaller to larger maps. This aspect is crucial for reinforcing the notion that a truly adaptive model should not only excel in environments it has been directly trained on but also demonstrate a high degree of transferability and performance across scenarios of different scales. To this end, we selected a range of map sizes similar to those used on our previous experiments – 8×8, 16×16, 24×24, 32×32, 48×48, and 64×64 – to thoroughly explore how our model navigates and adapts to varying levels of complexity. Given our emphasis on scalability, we ensured consistency in the features and obstacles present across all maps, differing only in their spatial dimensions, to isolate the effects of scale on the model's effectiveness. In all maps our agent starts at the upper-left corner of the map and must traverse an U-shaped path to reach the goal on the upper-right corner. In Figure 5.19, we can see the 8x8 map used on our experiments. All other maps present the same path, but in a different scale.

5.2.1 Encoder Generalization

In this exploration, our primary focus is on assessing the consistency of spatial feature representations generated by the SPP layer within our network across varying map scales. To this end, we initially trained our model on an 8×8 map for 30,000 steps, then guided the agent along a predetermined route across maps of different sizes. During this process, we captured and stored the SPP layer outputs for each state encountered along the trajectory in an array A, with each element a_i representing the SPP output at

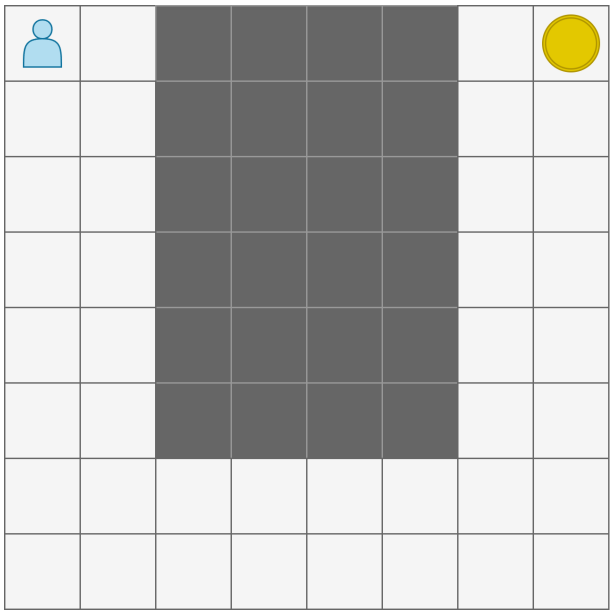

Figure 5.19: Illustration of the 8x8 grid utilized in our Frozen Lake experimentation: Starting from the upper-left corner, the objective of the agent is to navigate to the goal located at the upper-right corner. The grid features cells depicted in white, representing frozen terrain that is safe for the agent to traverse, contrasted with the dark cells, which represent holes to be avoided.

the i-th step. Our goal is to analyze and quantify the differences in representations for analogous states across varying map sizes.

Given that larger maps naturally lead to longer trajectories for the identical route, we implemented a uniformization process to facilitate a more equitable comparison. This process involves adjusting for map dimensions, using the 8×8 dimension as a standard reference. Specifically, utilizing dimension 8 as a reference, where each map expands by a factor of n, we aggregated the representations at intervals of n steps along the trajectory, computing the mean representation for these grouped states. This adjustment yields arrays of uniform length for each map, enabling us to then compute the Euclidean distance between the representation arrays of each map pair. The resulting distances can be seen in Table 5.2.

The results from our analysis reveal an intriguing pattern: the representations in larger maps demonstrated minimal variance, suggesting a high degree of similarity. The distances among representations of the 24×24, 32×32, 48×48, and 64×64 maps are all less than 5.2, with some of them as low as 1.4. Conversely, representations derived from smaller maps exhibited a greater deviation when compared to others. The greatest deviation registered was between the maps 8×8 and 24×24, with a value of 71. All the distances between the 8×8 and the other maps were bigger than 51, almost 10 times larger than what we observed between the larger maps. This indicates that as the map size increases, the model's ability to generate consistent representations improves, show-

Map	8x8	16x16	24x24	32x32	48x48	64x64
8x8	0.00	51.67	71.00	69.59	68.08	65.82
16x16	51.67	0.00	19.32	17.91	16.41	14.15
24x24	71.00	19.32	0.00	1.40	2.91	5.17
32x32	69.59	17.91	1.40	0.00	1.51	3.76
48x48	68.08	16.41	2.91	1.51	0.00	2.25
64x64	65.82	14.15	5.17	3.76	2.25	0.00

Table 5.2: Euclidean distance between the representation of different map sizes. The table displays the pairwise Euclidean distances among all maps, with rows and columns representing the map sizes (8×8, 16×16, 24×24, 32×32, 48×48, and 64×64). This comparison highlights the similarity between representations of bigger maps and dissimilarity of smaller maps.

casing a robustness in handling spatially complex environments. On the other hand, the increased divergence seen in smaller maps underscores potential challenges in generalizing from limited spatial contexts to more expansive scenarios. This suggests that the spatial features learned from smaller maps may not encapsulate enough variability, leading to less robust representations when scaled to larger environments. To address this issue, enhanced adaptation strategies or modifications in model architecture may be necessary.

5.2.2 Policy Transferability

In this subsection, we investigate the transferability of our model's policy across different map sizes. The policy is derived from the trained DQN by selecting the action with the highest Q-value for a given input state. We focused on comparing how well our model could transfer a policy learned on one scale to another. Initially, we trained the agent on the 8×8 map for 30,000 steps and evaluated its generated policy on every map. The results, shown in Figure 5.20, demonstrate that the agent learned an optimal policy on the 8×8 map, consistently reaching the goal by traversing the shortest path. However, when this policy was transferred to larger maps, its effectiveness diminished significantly. The model struggled to adapt and extend the policy to the 16x16 map. The transferred policy exhibits erratic and ineffective behaviors, indicating difficulty in scaling a policy acquired in a smaller environment to a larger one. For instance, if we trace the agent's trajectory on the 16×16 map starting from the top left corner (the initial state), we can observe that the agent quickly becomes trapped in a loop. This suboptimal transfer was unsurprising given the observed discrepancies in representations for smaller maps seen in the previous subsection.

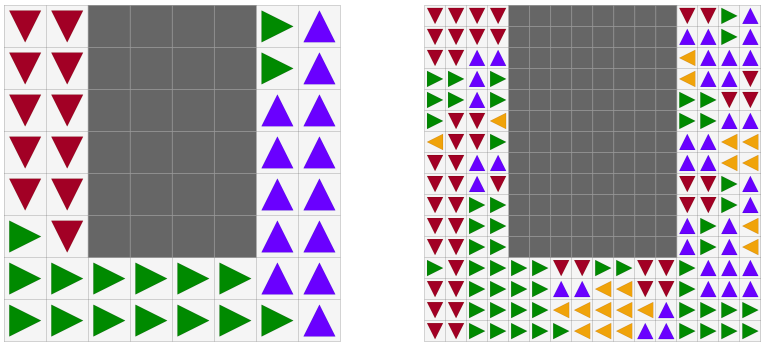

Figure 5.20: Agent policy transferability: on the left, the policy derived from training on the 8×8 map; on the right, the corresponding policy transferred to a 16×16 map. Significant discrepancies between the policies on each map are evident, indicating a suboptimal transfer.

To delve deeper into this phenomenon, we extended our investigation by conducting supplementary experiments. Here, we conducted additional experiments by training the agent on the 24×24 map and assessing its performance on larger maps. The resultant policies are illustrated in Figure 5.21. Remarkably, in this scenario, we observed a highly successful transfer, where most of the actions learned on the 24×24 map are transferred to the corresponding expected action on the the 48×48 map, indicating a robust ability of the model to generalize the policy from intermediate to larger spatial contexts. Although the resulting policy is not optimal and the agent still struggles to reach the goal, it is important to note that our primary focus in this section's experiments is on the agent's ability to generalize its learning across different map scales rather than its specific performance.

These contrasting outcomes shed light on the nuanced dynamics of policy transfer across different scales, emphasizing the critical role of scale considerations in Reinforcement Learning. The failure of the 8x8-trained policy to generalize to larger maps highlights the limitations of transferring knowledge acquired with small observation sizes to larger ones. On the other hand, the successful generalization from the 24x24 map to the 48x48 map underscores the importance of training on sufficiently complex environments. When trained on an intermediate-sized map, the agent encounters a diversity of spatial patterns and strategic challenges that better prepare it for even larger maps. This experience enables the model to develop more generalized and adaptable policies, capable of scaling to different sizes.

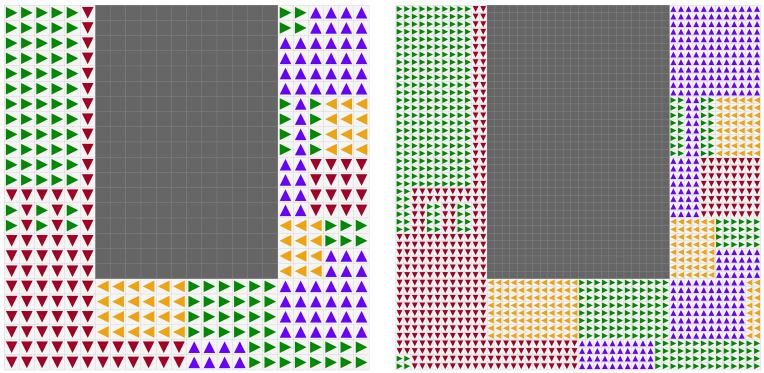

Figure 5.21: Agent policy transferability: on the left, the policy derived from training on the 24×24 map; on the right, the corresponding policy transferred for a 48×48 map. Despite the difference in map dimensions, the transferred policy closely matches the original policy, with only a few deviations.

5.2.3 Agent performance

Throughout our investigation, we observed a peculiar trend in our agent's performance. While it exhibited proficiency in learning the optimal policy within smaller maps (8×8 or smaller), it encountered significant challenges navigating larger maps to reach the goal. This issue was evident in the resulting policies from previous experiments: the agent trained on the 8x8 map successfully reached an optimal policy, whereas the agent trained on the 24x24 map struggled, failing to reach the goal consistently.

Interestingly, this issue is not unique to our model; we conducted tests employing other architectures and algorithms from established libraries like Stable Baselines [Raffin et al., 2021], only to find even worse results. These models struggled to learn even in the context of smaller maps. This predicament appears to stem from an inherent incongruity between the simplicity of the environment and the design focus of Deep Reinforcement Learning models, which are typically geared towards addressing more complex tasks. This is particularly relevant for environments like Frozen Lake, where the simplicity of the task does not fully leverage the advanced capabilities of DRL algorithms.

Despite these performance challenges, the significance of our analysis in this section remains intact. Our primary objective was to develop a scale-invariant DRL model and evaluate its ability to generalize learning across different map sizes rather than evaluating DQN efficacy in such environments. The insights gained from these experiments are valuable, especially when considering more complex environments where our approach did not display the same performance issues.

Chapter 6

Conclusion

6.1 Overview

In this thesis, we tackle the limitations of Reinforcement Learning (RL) frameworks that struggle when faced with varying sizes of state representations. We introduce a novel architecture that integrates Grid-wise [Han et al., 2019] Control with Spatial Pyramid Pooling [He et al., 2015], crafting a versatile model adept at learning from any grid-based setting without the need for structural modifications. This design allows for the smooth transfer of learned behaviors across similar environments. To enhance the model's adaptability, we have also devised a novel training methodology that encompasses a variety of environments, each with unique state representation dimensions. This method offers the flexibility to adjust the focus on specific environments based on the requirements of the task at hand.

Our experiments conducted within the Gym-μRTS environment [Huang et al., 2021], suggest that our model has potential advantages over existing state-of-the-art solutions in terms of efficiency and generalization capabilities. Despite these promising results, our findings should be viewed as preliminary, as they are based on a limited number of runs for each experiment. Under these conditions, our model demonstrated strong performance in certain instances, outperforming baseline models, particularly under constrained training scenarios. However, throughout our experiments, we encountered learning instabilities and generalization problems while working with small maps. Upon conducting an in-depth analysis within the Frozen Lake environment, we confirmed that our model struggles with very small maps, highlighting a significant area for improvement.

In summary, this research has taken initial steps toward developing a scale-invariant RL model with the potential to generalize across environments with diverse scale settings. While these results highlight the potential of our approach, further experimentation and refinement are necessary to fully establish its effectiveness. By addressing the identified limitations, we aim to further enhance the model's robustness and applicability, contributing to the development of more versatile and effective RL solutions for complex and varied

environments.

6.2 Future Work

Looking ahead, our focus extends to addressing the shortcomings found. Future work could explore alternative strategies to address the model's limitations in small maps, while simultaneously evaluating its effectiveness across a wider range of map sizes to ensure consistent performance. Furthermore, one could delve into alternative approaches for selecting environments within the refined PPO algorithm, such as implementing a weighted random selection mechanism, to gain deeper insights into its impact on the model's learning trajectory. Additionally, investigating other scale-invariant architectures like Graph Attention Networks (GATs) or transformer-based approaches could prove beneficial. These approaches have shown promise in handling varying input sizes and capturing long-range dependencies, which may help overcome the current model's limitations.

References

Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. Deep rts: a game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE, 2018.

Nicolas A Barriga, Marius Stanescu, and Michael Buro. Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Transactions on Games*, 10(1):69–77, 2017.

Nicolas Arturo Barriga, Marius Stanescu, and Michael Buro. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Jay Burmeister and Janet Wiles. The challenge of go as a domain for ai research: a comparison between go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, pages 181–186. IEEE, 1995.

Michael Buro. Real-time strategy games: A new ai research challenge. In *IJCAI*, volume 2003, pages 1534–1535, 2003.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in starcraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for rts game combat scenarios. In *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*, volume 8, pages 112–117, 2012.

B Jack Copeland. *The essential turing*. Clarendon Press, 2004.

Stefan Edelkamp and Richard E Korf. The branching factor of regular search spaces. In *AAAI/IAAI*, pages 299–304, 1998.

Daniel James Edwards and TP Hart. The alpha-beta heuristic. 1961.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Lei Han, Peng Sun, Yali Du, Jiechao Xiong, Qing Wang, Xinghai Sun, Han Liu, and Tong Zhang. Grid-wise control for multi-agent reinforcement learning in video game ai. In *International Conference on Machine Learning*, pages 2576–2585. PMLR, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, 37(9):1904–1916, 2015.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. In Roman Barták, Fazel Keshtkar, and Michael Franklin, editors, *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, Hutchinson Island, Jensen Beach, Florida, USA,*

May 15-18, 2022, 2022. doi: 10.32473/flairs.v35i.130584. URL https://doi.org/10.32473/flairs.v35i.130584.

Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning. In *2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, August 17-20, 2021*, pages 1–8. IEEE, 2021. doi: 10.1109/CoG52621.2021.9619076. URL https://doi.org/10.1109/CoG52621.2021.9619076.

Vince Jankovics, Michael Garcia Ortiz, and Eduardo Alonso. Efficient entity-based reinforcement learning. *arXiv preprint arXiv:2206.02855*, 2022.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

Julian RH Marino, Rubens O Moraes, Tassiana C Oliveira, Claudio Toledo, and Levi HS Lelis. Programmatic strategies for real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 381–389, 2021.

Maja J. Mataric. Reward functions for accelerated learning. In *Machine Learning, Proceedings of the Eleventh International Conference*, 1994.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

Leandro C Medeiros, David S Aleixo, and Levi HS Lelis. What can we learn even from the weakest? learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7761–7769, 2022.

Marvin Minsky and Seymour A Papert. *Perceptrons, Reissue of the 1988 Expanded Edition with a new foreword by Léon Bottou: An Introduction to Computational Geometry*. MIT press, 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013a.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013b. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

Santiago Ontanón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4): 293–311, 2013.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, 22:268:1–268:8, 2021.

Benjamin Rolf, Ilya Jackson, Marcel Müller, Sebastian Lang, Tobias Reggelin, and Dmitry Ivanov. A review on reinforcement learning algorithms and applications in supply chain management. *International Journal of Production Research*, 61(20):7151–7179, 2023.

Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

Arthur L Samuel. Some studies in machine learning using the game of checkers. ii—recent progress. *IBM Journal of research and development*, 11(6):601–617, 1967.

Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The Starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*, 2019.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

Peter Shotwell. The game of go: Speculations on its origins and symbolism in ancient china. *Changes*, 2008:1–62, 1994.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems*, 30, 2017.

Alan M Turing. *Computing machinery and intelligence*. Springer, 2009.

Alberto Uriarte and Santiago Ontanón. Game-tree search over high-level game states in rts games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 10, pages 73–79, 2014.

J v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Chao Wang, Jian Wang, Yuan Shen, and Xudong Zhang. Autonomous navigation of uavs in large-scale complex environments: A deep reinforcement learning approach. *IEEE Transactions on Vehicular Technology*, 68(3):2124–2136, 2019.

Xiangjun Wang, Junxiao Song, Penghui Qi, Peng Peng, Zhenkun Tang, Wei Zhang, Weimin Li, Xiongjun Pi, Jujie He, Chao Gao, et al. Scc: an efficient deep reinforcement learning agent mastering the game of starcraft ii. In *International Conference on Machine Learning*, pages 10905–10915. PMLR, 2021.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

Georgios N Yannakakis and Julian Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, 2014.

Won Joon Yun, Sungwon Yi, and Joongheon Kim. Multi-agent deep reinforcement learning using attentive graph neural architectures for real-time strategy games. In *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2967–2972. IEEE, 2021.

Albert L Zobrist. A model of visual organization for the game of go. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 103–112, 1969.

Albert Lindsey Zobrist. *Feature extraction and representation for pattern recognition and the game of Go*. The University of Wisconsin-Madison, 1970.