Component-Action Deep Q-Learning for Real-Time Strategy Game AI

by © Richard Kelly A Thesis submitted

to the School of Graduate Studies in partial fulfillment of the

requirements for the degree of

Master of Science, Department of Computer Science, Faculty of Science

Memorial University of Newfoundland

May, 2021

St. John's Newfoundland and Labrador

Abstract

Real-time Strategy (RTS) games provide a challenging environment for AI research, due to their large state and action spaces, hidden information, and real-time gameplay. The RTS game StarCraft II has become a new test-bed for deep reinforcement learning (RL) systems using the StarCraft II Learning Environment (SC2LE). Recently the full game of StarCraft II has been approached with a complex multi-agent RL system only possible with extremely large financial investments. In this thesis we will describe existing work in RTS AI and motivate our work adapting the deep Q-learning (DQN) RL algorithm to accommodate the multi-dimensional action-space of the SC2LE. We then present the results of our experiments using custom combat scenarios. First, we compare methods for calculating DQN training loss with action components. Second, we show that policies trained with component-action DQN for five hours perform comparably to scripted policies in smaller scenarios and outperform them in larger scenarios. Third, we explore several ways to transfer policies between scenarios, and show that it is a viable method to reduce training time. We show that policies trained on scenarios with fewer units can be applied to larger scenarios and to scenarios with different unit types with only a small loss in performance.

Acknowledgements

I wish to thank my supervisor, Dr. David Churchill, for his guidance through completing this thesis, and for encouraging a fun and engaging environment for both undergraduate and graduate students to learn and conduct research on AI and games. Thank you as well to the other students of the lab.

I'm also very grateful for the support of my family and especially my wonderful partner, Gabriela, throughout my time as a student.

Contents

List of Tables

List of Figures

List of Abbreviations

A2C Advantage Actor Critic. 13, 16, 53, 94

A3C Asynchronous Advantage Actor Critic. 13, 16

AI artificial intelligence. 1–7, 9–11, 14–19, 40, 42, 45, 46, 60, 61, 67, 68, 73, 84, 93, 94

ALE Atari Learning Environment. 12, 13, 33, 34, 38, 60, 94

ANN artificial neural network. 19

CNN convolutional neural network. 12, 19, 23, 24, 38, 41, 56, 57, 81

CTL curriculum transfer learning. 15, 17, 87, 89, 92, 94

DQN deep Q-network. 7, 8, 12, 13, 15, 18, 25, 33–38, 40, 46, 47, 49, 50, 52, 54, 57, 60, 67, 68, 70, 91, 93, 94

LSTM long short-term memory. 13

MCTS Monte Carlo tree search. 11, 13, 68

MDP Markov decision process. 27, 28

ML machine learning. 1, 5, 6, 25, 40

MSE mean squared error. 21, 34, 52, 53

RL reinforcement learning. 1, 4, 5, 7, 8, 12–19, 21, 25, 26, 28–30, 38, 40, 41, 46, 47, 49, 60–63, 67–69, 78, 91, 93–95

RNN recurrent neural network. 13, 14, 50, 51

RTS real-time strategy. 1–7, 9–12, 14–16, 18, 45–47, 63, 68, 69, 91, 93, 94

SC2LE StarCraft II Learning Environment. 7, 8, 15, 16, 40, 94

SGD stochastic gradient descent. 22, 34

TD temporal difference. 25, 30, 31, 33, 34, 36–38, 52, 53, 67, 70

Chapter 1

Introduction

Games have been used as benchmarks for the progress of artificial intelligence (AI) since the early days of AI research [31, 35]. Games have the benefit of being easily simulated, they have well-defined limits on which actions can be taken, and the AIs that play them are easy to compare to human ability. The games targeted by AI researchers are naturally of increasing complexity and require more advanced techniques to master. The Checkers program Chinook defeated the world champion player Marion Tinsley in 1994 using heuristic search and a database of end-game positions [33]. Chess champion Garry Kasparov was defeated by IBM's Deep Blue chess playing program in 1997, making use of heuristic search, parallel computing, and a Grandmaster game database [3]. Google DeepMind achieved expert human-level play in a suite of 49 classic Atari video games using a single deep reinforcement learning (RL) algorithm in 2015 [26]. The world champion Go player, Lee Sedong, was defeated by Google DeepMind's program AlphaGo in 2016 [38], using a combination of heuristic search, deep learning, and RL. In recent years the video game genre of real-time strategy (RTS) games have become a popular testbed for research in Artificial Intelligence, due to the complex problems they present for players, and newer techniques such as deep RL are now easier to apply to this domain with the release of new machine learning (ML) APIs for RTS games.

1.1 Real-Time Strategy Games

RTS games, in addition to being played in real-time and involving strategy (attributes which many games have), have a number of other elements that are typical in the genre. They are usually military or combat themed games, in which 2 or more players control multiple units and compete to gather resources and destroy the other side(s). The StarCraft series of games is a popular example, being the focus of much AI research and competitive human play [5]. StarCraft is a science-fiction themed RTS game featuring 3 asymmetrically structured races, the Terran (humans), Protoss, and Zerg. In StarCraft and most RTS games, players control units on a battlefield (map) from an overhead perspective. Figure 1.1 shows the main elements of the StarCraft II interface, including the map and *minimap*, which shows a view of the entire map for quickly gleaning information about the game state and for navigation (a player has to choose which area of the map to focus on at any given time). Units, including workers and various types of combat units, can be given orders to move anywhere on the map, harvest resources, and attack enemy units. Players also control and build buildings which are used to research upgrades and build more units. In the StarCraft series a player starts with a *base* and 4 workers, and will eventually expand to build more bases near other resource locations as their economy grows. The goal in a standard 2-player (1 vs. 1) game in StarCraft is to destroy all of the other player's buildings. At the large (macro) scale, players must balance spending resources on expanding their economy, defending their buildings, and attacking the enemy's buildings. At a smaller scale, players can micro-manage individual aspects of the game, most notably combat, where formations and use of specific abilities by units is important. Successful play depends on long-term planning as well as the ability to change strategies in reaction to opponent play. Most RTS games feature a "fog of war", as shown in Figure 1.2, meaning that a player can't see enemy units in areas of the map in which they don't currently have units. This element adds to the complexity of the game, requiring tactics such as scouting to find the location and type of enemy buildings and units.

Games in the RTS genre are complex, with many features that are closer to real-world

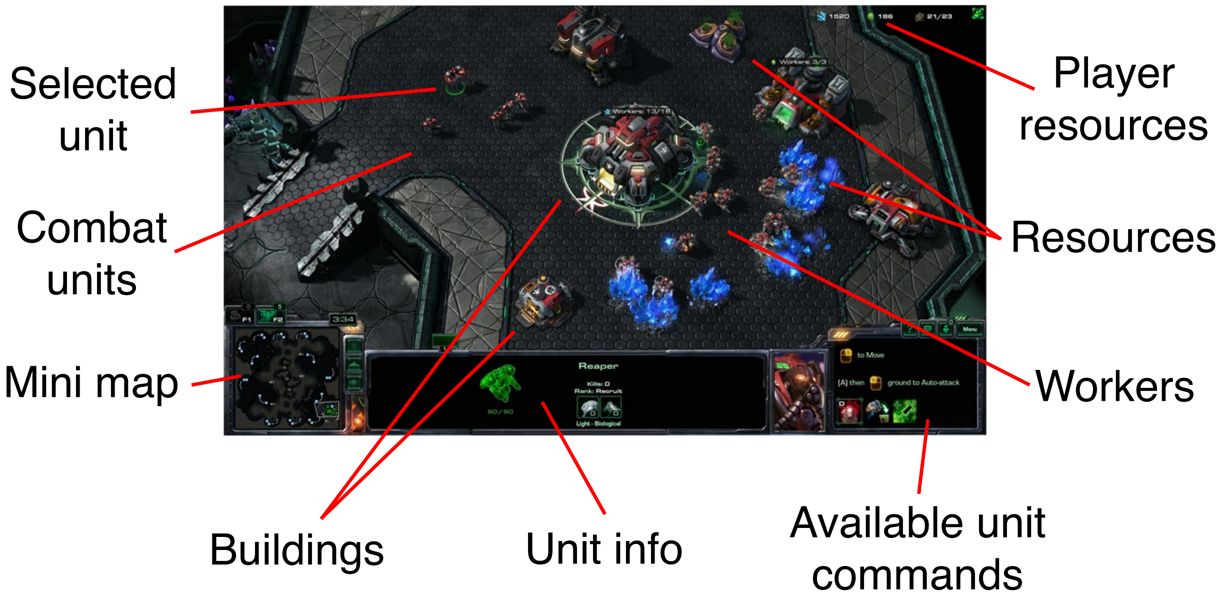


Figure 1.1: A screenshot from a 1 vs. 1 StarCraft II match with labelled user interface elements.

problems than previous board and video board game AI benchmarks. RTS games are usually played in real-time, cannot be modelled for use in search algorithms, and have much larger state and action spaces. One indication of a game's complexity is the number of possible game states. Chess has about 10^{50} states, Go 10^{70} states, and the RTS game StarCraft is estimated to have a lower bound of over 10^{1685} states [29]. Other factors also contribute to the complexity of a game. For example, there are 20 possible opening moves in a game of Chess, and 361 in Go (though many positions are equivalent for the first few moves of the game due to board symmetry). In a commercial RTS game there are thousands of valid positions to move a unit to. Additionally, games in the RTS genre are imperfect information games, meaning that all the information about the game state is not available to a player.

The real-time nature of RTS games means that players require dexterity to play the game well, whereas turn-based board games such as Chess have no such requirement. Skilled RTS players take actions using the mouse and also many memorized keyboard shortcuts, allowing them to take up to hundreds of actions per minute (APM) [49]. For an AI controlled player, as with a human, real-time play means that there is a time limit for short-term decision

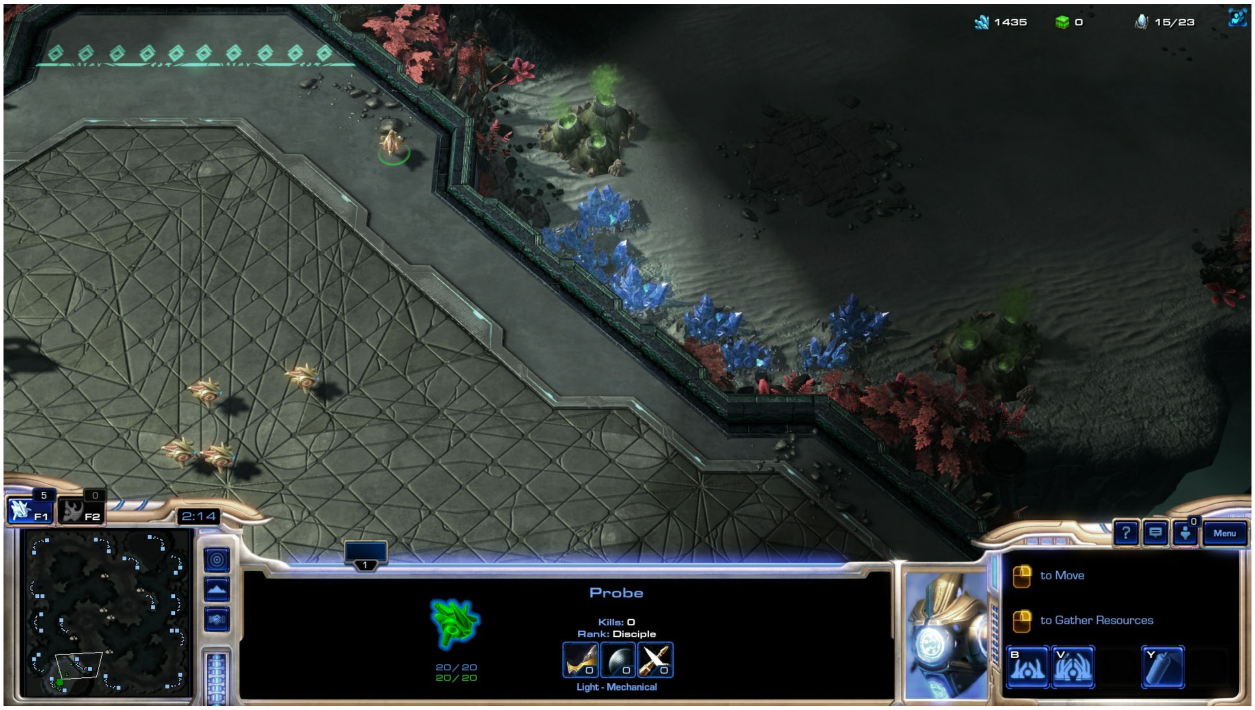


Figure 1.2: A screenshot from StarCraft II showing the fog of war on the right side.

making. However, AI players have an advantage because they can issue actions as fast as the computer allows inputs, whereas human players are limited in speed and dexterity.

1.2 Motivation

RTS games provide interesting environments for AI research, leading to many competitions and APIs for working on existing RTS video games. Recently, deep RL has become a leading approach to solving the problems of playing RTS games, prompting Google DeepMind to call the RTS game StarCraft II a "new grand challenge for reinforcement learning" [51].

1.2.1 Real-Time Strategy Game AI

RTS games contain complex sub-problems providing many algorithmic challenges. Planning is made difficult by the large state and action spaces in RTS games. Additionally, planning needs to take place on multiple time-scales, from short term for managing battles, to long-term for managing economy and build order (the order in which buildings, upgrades, and units should be acquired). RTS AI agents (bots) can learn from prior games against an opponent, or learn in-game through opponent-modelling. Uncertainty due to the partial observability of RTS games requires good knowledge representation by AI agents. Spatial reasoning is important in many tasks such as building placement, unit formations, and strategic placement of units across the map. Domain knowledge exploitation in RTS games has included hard-coded strategies in bots such as opening build orders, as well as mining data from human game replays. [29]

The availability of multiple RTS game APIs [12, 28, 43, 51] and bot competitions [4, 28] has made the game genre an ideal environment for testing novel AI methods. One of the most popular RTS games for research is currently StarCraft: Broodwar, released by Blizzard Entertainment in 1998. Using the Broodwar Application Programming Interface (BWAPI) [12] released in 2009, many bots for StarCraft: Broodwar have been developed and have competed in several annual competitions and ladders. Until recently the state-of-the-art in StarCraft bots was a mixture of AI techniques such as heuristic search and spatial reasoning, combined with scripted rules based on expert knowledge [7]. These bots can be defeated by skilled human players who can recognize and exploit the strategies used by the bots. At the 2017 CIG (Computational Intelligence in Games) conference, the top StarCraft bots from the conference's tournament were matched against a professional player who won all matches [4]. The bots were able to defeat a novice player and a "middle-level" player.

1.2.2 StarCraft II AI with Deep Learning

In 2018, Google DeepMind unveiled an AI agent called AlphaStar [49], which used ML techniques to play StarCraft II at a professional human level. AlphaStar's neural network was initially trained using supervised learning from hundreds of thousands of human game traces, and then continued to improve via self-play with RL, a method by which the agent improves its policy by learning to take actions which lead to higher rewards more often. While this method was successful in producing a strong agent, it required a massive engineering effort, with a team comprised of more than 30 world-class AI researchers and software engineers. AlphaStar also required an enormous financial investment in hardware for training and game play, using the equivalent of over 50000 CPU cores to run simultaneous instances of StarCraft II and over 1200 Tensor Processor Units (TPUs) to train the networks, as well as a large amount of infrastructure and electricity to drive this large-scale computation [50]. While AlphaStar is estimated to be the strongest existing RTS AI agent, it does not yet play at the level of the world's best human players (it achieved the highest rank possible in a human ladder, but did not participate in a tournament with any of the world's top players). The creation of AlphaStar demonstrated that using deep learning to tackle RTS AI is a powerful solution, however applying it to the entire game as a whole is not an economically viable solution for anyone but the worlds largest companies.

In 2017, Blizzard (developer of the StarCraft games), released SC2API: an API for external control of StarCraft II. DeepMind, in collaboration with Blizzard, simultaneously released the SC2LE with a Python interface called PySC2 designed to enable ML research with the game [51]. AlphaStar was created with tools built on top of SC2API and SC2LE. PySC2 allows commands to be issued in a way similar to how a human would play; units are selected with point coordinates or by specifying a rectangle the way a human would with the mouse. Actions are formatted as an action function (e.g. move, attack, cast a spell, unload transport, etc.) with varying argument counts depending on the function. This action representation differs from that used in other RTS AI research APIs, including the TorchCraft ML library for StarCraft: Broodwar [43]. Representing actions as functions with parameters creates a complicated action-space that requires modifications to existing RL algorithms.

1.3 Research Questions

In this thesis we explore cost-effective solutions for learning policies for RTS games, using PySC2 and StarCraft II as a testing environment. One possible solution is to use the idea of transfer learning: learning to generate policies for sub-problems of RTS games, and then using those learned policies to generate actions for many other sub-problems within the game, which can yield savings in both training time and infrastructure costs. Deep RL, which is RL using neural networks to learn policies and state value, has been shown to be successful in RTS games and other domains. We apply modifications to an existing RL algorithm and apply it to this problem.

Specifically, we seek to determine:

1. Can we train a model on one StarCraft II combat scenario and then apply it to another scenario, which differs in some non-trivial way such as size or unit type, and achieve similar performance.

2. How suitable is the deep Q-network (DQN) RL algorithm to this type of problem when modified to use an action space similar to human-control.

1.4 Thesis Outline

This thesis is organized as follows: in Chapter 2 we summarize previous work in RTS game-playing AI and work using deep RL for playing games. In Section 2.3 we focus on deep learning for RTS games and specifically in the StarCraft II domain using the StarCraft II Learning Environment (SC2LE). In Chapter 3 we present important concepts used in our research, deep learning and RL, and define terms that will be used throughout the rest of the thesis. In Chapter 4 we describe our methodology. In Section 4.1 we detail how we use the SC2LE for our experiments and in Section 4.2 we describe our work on another RTS environment, microRTS. In Section 4.3 we describe our implementation of component-action

DQN, which modifies the DQN RL algorithm to work with the function-parameter action representation used in the SC2LE. In Section 4.4.3 we list custom StarCraft II combat scenarios developed for our experiments. In Chapter 5 we detail the experiments we ran testing the performance of component-action DQN in various scenarios and present the results of those experiments. In Chapter 6 we conclude by summarizing the work presented in this thesis and promising areas for future research.

Chapter 2

Previous and Related Work

RTS game AI research aims to create complete game-playing agents that are comparable to humans in effectiveness. Work in RTS AI has been ongoing since the early 2000s, picking up with the release of the first API for a commercial RTS game in 2009 and subsequent bot tournaments starting in 2010 [29]. We list several previous works in RTS AI which don't use deep learning in Section 2.1. Meanwhile, deep learning has been applied to various board and video game environments, ranging from classic 2D games to first-person shooters [18], which we describe in Section 2.2. In Section 2.3 we discuss deep learning methods which have been applied to RTS games.

2.1 Real-Time Strategy Game AI and Bots

RTS game playing bots historically have relied on a mix of scripted behaviours and other AI methods [7]. Scripted behaviours are rule-based systems where the rules are programmed or developed by humans with expert knowledge. For example, a rule could be that when a bot first encounters invisible enemy units, it begins to build "detector" units which can see invisible enemies. Bots that rely on highly scripted behaviours can be powerful when successfully executing those strategies against an unprepared opponent, but can easily be countered by a skilled human player that reacts to the bot's strategy. Search-based techniques exist for many areas of RTS gameplay, but are impractical for playing a full RTS game.

2.1.1 Scripted Bots

Most commercial RTS games are programmed with relatively simple built-in AI opponents for humans to play against, with a range of difficulty levels. These systems are often implemented as hierarchical finite state machines (FSM) [29]. Often higher difficulty versions receive benefits such as extra resources to make them harder to play against without increasing the complexity of the AI system. Built-in AI opponents are also not constrained by the fog of war, though in some games they will send a scouting unit which appears to discover the player so that the AI opponent appears to be playing intelligently.

Most bots that compete in StarCraft competitions use some amount of scripted behaviour. Opening build orders, giving the bot a precise schedule for when to produce which units and buildings in the opening minutes of a match, are common. In some bots subsequent build orders are generated by following simple rules. Seeing an enemy unit with a certain trait (e.g. invisible, flying, etc.), may cause a counter-unit to be built, such as a unit that can detect invisible enemies. Another element that is often scripted is the timing of expanding to a second base. The placement of buildings requires some terrain analysis, but often specific patterns are scripted, such as placing buildings at a choke point to block enemy units from entering a base. Elements of combat may be scripted, such as where to attack based on the location of the enemy bases. Most bot authors use domain knowledge to divide gameplay into individual tasks that can be handled by different AI techniques [7].

2.1.2 Search and Other Methods

Search is used in many ways in RTS bots. The built-in AI of the games use search for pathfinding, which extends to a player's own units if they are given an order to move. Bots (as well as human players) may override that pathfinding by giving units order to move shorter distances more often. Pathfinding and building placement are supported by terrain analysis performed at the beginning of a match or in advance per map. Potential fields and influence maps are also used for avoiding enemies during pathfinding [29]. Heuristic search is applied to build orders and combat micro-management in StarCraft with the Build-Order Search System (BOSS) and the SparCraft combat simulator, respectively, in *UAlbertaBot* [5]. BOSS finds build orders for arbitrary build goals and can spread the computation time over multiple game frames. For a build goal such as "produce 10 tanks with the siege-mode upgrade", the build order would need to decide when to produce more workers to increase the rate of income generation, when to start building a factory for the tanks, when to start researching the siege-mode upgrade, and when to build each tank. SparCraft runs many combat simulations to determine if a group of units will win a battle or not, causing them to retreat or advance. The StarCraft bot *LetaBot* uses Monte Carlo tree search (MCTS) for large scale unit group movement [4]. In [46] combat models are learned from replay data and used in MCTS to predict combat outcomes in RTS games. Search methods such as MCTS and alpha-beta are applied to the full game of microRTS, a simple RTS game designed for AI research [27]. Search approaches have not been applied to the full game of StarCraft as one search problem due to the high number of states and actions [29].

Many StarCraft bots select from among a list of strategies for each match based on past matches against a specific opponent, using UCB1 or similar formulas to select the next strategy to use. Some bots have a hierarchy of strategy choices, for example early, mid and late-game strategies [4]. Several StarCraft Bots approach playing the game as a multi-agent problem, having each unit act independently based on scripted rules or other methods, sometimes with some influence over other nearby units or long-term strategy. *ForceBot*'s units have rule's based behaviour. *Iron*'s units can switch between 25 behaviours, and certain units are able to recommend spending resources. *PurpleWave* groups units into squads using nearest neighbour clustering and simulates battles using the squads. Individual choices are still made by the units [4].

2.2 Deep Learning for Game AI

Deep learning, including supervised learning, unsupervised learning, and various reinforcement learning algorithms have been applied to board games [38, 39] and numerous video games [18] in recent years. Since 2014, deep RL has been used in a number of game genres, including arcade games, racing games, first-person shooters, open-world games, RTS games, and general game playing, as well as other domains such as robotics [18, 2]. Deep RL especially has been applied to many areas with many new algorithms building off of previous ones released since 2014 [18]. Neural network architectures also vary with the games and corresponding problems these algorithms are applied to.

In 2013 Mnih et al., with Google DeepMind, introduced a new reinforcement learning algorithm called DQN for playing Atari 2600 games [25]. In results published in 2015, their single learning algorithm and CNN-based network architecture was able to learn to play 49 different Atari games at expert human level using only raw screen pixels and score as input, using the Atari Learning Environment (ALE) [26]. The network was independently trained for each game from random initialization, showing that one network architecture and learning algorithm could be generalizable to different games with completely different gameplay mechanics. The neural network used for playing Atari from pixels uses a convolutional neural network (CNN), taking advantage of advances in image recognition using CNNs. Actions in ALE are selected from among 18 discrete choices based on an Atari joystick controller. The DQN Atari agent uses the last four frames of the game as input to better represent the game state (e.g. to capture the direction of movement of figures on the screen). The games that this method did not produce good results for were those for which strategies with long-term planning were required, such as the game Montezuma's Revenge, in which the player solves puzzles by finding keys and bringing them to other rooms in the game. These games don't show the complete game state on the screen at one time, which is challenging for a learning system with no memory beyond a few frames of the game.

A number of other algorithms building off of DQN have been applied to the ALE with better performance. Notably, enhancements to DQN including prioritized experience replay [34], which improves the selection of recorded training data to learn from, and double DQN [48], which improves the stability of DQN. Those enhancements were later combined with noisy networks [9], which change the exploration method, into a single algorithm called Rainbow [13], which achieved higher performance than any of the other DQN methods alone. The popular and related RL algorithms Asynchronous Advantage Actor Critic (A3C) and its synchronous counterpart Advantage Actor Critic (A2C) have also been applied to the ALE, achieving better results than the original DQN with shorter training times using CPUs rather than GPUs for training [24]. In this thesis we use a version of DQN with some enhancements and modified to handle the PySC2 action space, which is more complicated than that of the games in the ALE.

In 2016 Silver et al., also with Google DeepMind, published their new approach to playing the board game Go, AlphaGo, using supervised learning of expert human games, reinforcement learning through self-play, tree search, and deep neural networks; it was successful in defeating the European and later world champion human Go players [38]. This is an example of using convolutional neural networks for learning policy and evaluation functions while preserving the spatial information present in the game board state. In 2017 a major improvement, AlphaGo Zero, was announced, which surpasses Alpha Go (100 - 0 win rate) while using only self-play reinforcement learning with MCTS and no expert human knowledge [39]. AlphaGo significantly improves on the action space under consideration compared to the 18 actions used in the ALE, since Go is played on a 19 by 19 grid (meaning there are 361 actions available at the start of a game).

In [22] a recurrent neural network (RNN) using a long short-term memory (LSTM) component is applied to reinforcement learning in the first-person shooter (FPS) game DOOM. In FPS games the player's screen represents the viewpoint of a person moving around a game arena. If the player turns to look in one direction, it would be helpful to remember what was on the screen earlier, in another direction, or around a corner that was previously in view.

RNNs are neural networks that have a memory component which carries over information from previous input to the network. This work is an example of using an RNN to learn to play a game where parts of the state are hidden from the agent. This agent also used two networks that learn different parts of the game (movement/exploration and combat) separately, which are called on depending on the current game state. In this thesis we experiment with small scenarios in which the entire game state is available to the RL agent at once, so we don't use an RNN component.

2.3 Deep Learning in RTS Games

RTS game AI problems have been approached using neural networks with both supervised learning and RL. RL has been used both in multi-agent approaches, where one or more RL models are trained to independently give orders to multiple units, as well as in single-agent approaches.

2.3.1 Supervised Learning

Supervised learning is used in RTS research typically by examining human replay data and using that to make decisions for some subsection of the game. Stanescu et al. describe a CNN for state evaluation (predicting the winner of a game based on a non-terminal state) in microRTS, a simple RTS testbed for AI techniques, and compare its performance to other state evaluation techniques for both predicting outcomes and for selecting actions in play using tree search [40]. Search can be used for microRTS because of it's small state and action spaces compared to commercial RTS games. In [19] a neural network was trained on expert-level human replay data to learn macromanagement decisions (long-term strategy decisions) for StarCraft: Brood War. AlphaStar uses input from expert human replays to guide build orders as a component of its RL system [49]. In this thesis we don't use supervised learning or use any human replay data in our RL system.

2.3.2 Reinforcement Learning Approaches

Several works have looked at combat micromanagement in StarCraft: Brood War and StarCraft II as a multi-agent problem to be solved with deep learning. [32] introduced a platform using the SC2LE called the StarCraft Multi-Agent Challenge (SMAC), used for testing multi-agent RL techniques by treating each unit as a separate agent. Most of these approaches narrow the state-space by only considering the area within some radius of a unit to inform the policy for that unit. In [36], individual units choose one of two actions (attack or retreat) using a neural network-backed RL system, trained on the spatial data centred around a single military unit. In [47] a new reinforcement learning algorithm, zero order (ZO), is introduced, and results for ZO, DQN, and other RL algorithms are provided on some micromanagement combat mini-games. That work uses the Torchcraft library [43], which facilitates machine learning in StarCraft: Brood War using the Torch machine learning library.

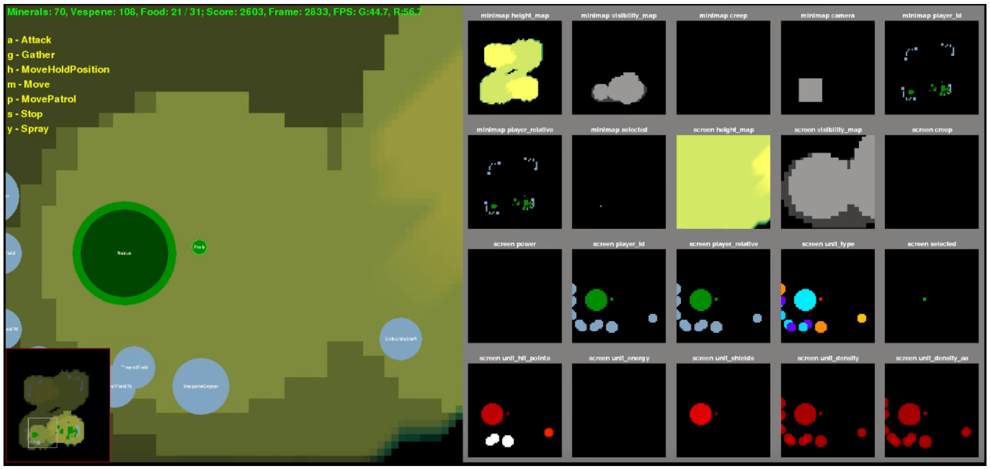
Multi-agent RL is combined with curriculum transfer learning (CTL) in several works to build up the abilities of agents. In [37] RL was used with CTL to train agents that represent individual unit types which are able to defeat the built-in AI. During training each unit of the same type shares the same network parameters and policy. The authors use CTL by training the same network on increasingly complex combat scenarios (e.g. more units and different types). The authors also use reward shaping to address the problem of delayed and sparse rewards, applying a -10 reward whenever a unit dies, and giving an immediate reward equal to damage inflicted when a unit makes an attack. In [15], a type of CTL is used in which the agents are trained using the state-action data from their opponents (rule-based AI players) in the first stage, followed by an RL stage of improving their policy by playing against the built-in AI. While treating combat as a multi-agent problem is shown to be effective in these works, this approach is very different to how a human player would play a full RTS game, and doesn't address the larger strategic planning required to master an RTS game. In this research we use the PySC2 action space to play the game as a single agent, similar to how a human plays.

RL has also been applied to the RTS AI testbed microRTS. Huang et al. used the microRTS environment to test different state representations for neural network input [16]. They trained an agent using the A2C algorithm, a synchronous version of the A3C policy gradient method [24], and experimented with using either a state input centred on each unit to give an order to that unit, or using a global state input of the entire map when ordering units. When the state input is local to the unit, the approach is multi-agent, with each unit receiving orders independently of the others. When the state input is global the single agent has to choose which unit to issue orders to. In both configurations the authors use action components which are combined into a complete action. This thesis uses a similar component-based action representation.

2.3.2.1 Reinforcement Learning Approaches to the Full Game of StarCraft II

Since the release of the SC2LE and PySC2 several works have been published using this environment with deep RL. Most notable is AlphaStar, developed by DeepMind, which achieved human professional level play with an extraordinary amount of training resources. Other approaches focus on ways to make the state or action space smaller. Figure 2.1 shows a visualization of the PySC2 state representation.

Several works have explored reducing the action-space by combining StarCraft II actions in PySC2 into a smaller number of *macro actions*. A bot called *TStarBot1* was trained using a reduced action space of 165 macro actions, which combine the actions available in PySC2 to make the atomic actions more meaningful and less numerous [41]. This strategy made RL training to play the full game against the built-in AI achievable. The bot could sometimes win against human players, though they quickly learn to adapt to the single strategy TStarBot1 learned. Macro actions have also been used in combination with a hierarchical RL architecture of sub-policies and states and curriculum transfer learning to progressively train an agent on harder levels of the built-in AI to learn to play the full game of StarCraft II [30]. In this work, macro actions, state representation, and reward functions


Minerals: 70, Vespene: 108, Food: 21 / 31; Score: 2603, Frame: 2833, FPS: G:44.7, R:56.7

a - Attack
g - Gather
h - MoveHoldPosition
m - Move
p - MovePatrol
s - Stop
y - Spray

Figure 2.1: Visualization of DeepMind's Python API state observation in the StarCraft II Learning Environment

(a combination of win/lose/tie and score) are independent to each sub-policy and a controller network (the controller chooses which sub-policy to use for a time period). Macro actions were generated by data mining human replays. CTL is used by training the agent on the easiest built-in AI difficulty, followed by progressively harder settings. In this thesis we do not use macro actions, but we do limit actions to only those relevant to the particular scenarios we use for training.

In [44] an agent was trained to learn unit and building production decisions (build order) using RL while handling resource harvesting and combat with scripted modules. The authors used versions with and without convolutional layers with input from the game's minimap; the version with the convolutional part performed better in tests against the built-in AI. DeepMind's AlphaStar agent for playing the full game of StarCraft II uses convolutional layers and recurrent modules to take advantage of spatial nature of the game state data and to learn sequences of actions, respectively. It uses build order information from human replays to bootstrap that part of learning. It also uses a novel league system to train multiple agents and uses previous versions as goals to overcome, ensuring that the final agent is minimizes exploitability to any previously generated agents [49].

Each of these recent works, except for AlphaStar, has reduced the scope of the RL problem in some way to a more manageable level in order to achieve some success in the full game of StarCraft II against the built-in AI. Notably, these other works have not used the complex action-space directly provided by PySC2, which mirrors how humans play the game. AlphaStar uses a modified version of the PySC2 action space, allowing the agent to select up to hundreds of units on-screen at a time (without being constrained to a rectangular mouse selection or using a saved "control group" as a human would) and issuing a command to them in one action. In this thesis we use a similar action space, with the additional constraint that selection of units is an action which must be taken first before issuing commands to those units with another action.

2.3.3 Our Approach

For our experiments we leverage several previous works. We use the PySC2 environment, developed by DeepMind [51], as our experimental environment. We use the DQN [26] RL algorithm, with the prioritized experience replay [34] and double DQN [48] enhancements. DQN has been successful in previous game environments, making it a good choice to expand upon. We use a component-action system, meaning that actions are composed of several components which are combined to make up a full action. Techniques similar to our implementation have been used in several other RTS AI works [16, 51]. In our experiments we use small custom StarCraft II combat scenarios with to test the effectiveness of our algorithm. Similar scenarios, pitting equal groups of units against each other, are used in other RL RTS works [47, 32]. We also use reward functions in our scenarios, which guide the learning of our RL agent, which are derived from RTS search heuristic evaluation functions [5].

Chapter 3

Background

In this chapter we explain the key concepts behind the tools that we use to implement Component-Action DQN for StarCraft II, deep learning and RL. Both deep learning (under different names) and RL have been used for decades [10, 42], but recently RL has emerged as a major technique for game AI when combined with deep learning (deep RL) [18].

3.1 Deep Learning

Deep learning refers to the use of artificial neural network (ANN)s with particularly "deep" or large networks, to learn to approximate functions by training on examples and then predict outputs on new data. Versions of ANNs have existed for many years, but have moved into common use along with the term "deep learning" since the mid 2000s [10]. In this research we make use of standard fully connected, or dense, layers and convolutional layers (CNNs).

3.1.1 Artificial Neural Networks

ANNs are function approximators inspired by neurons in the human brain. They are essentially a directed graph of real-numbered values organized into layers through which mathematical operations are performed. They take a number of input values and output one or

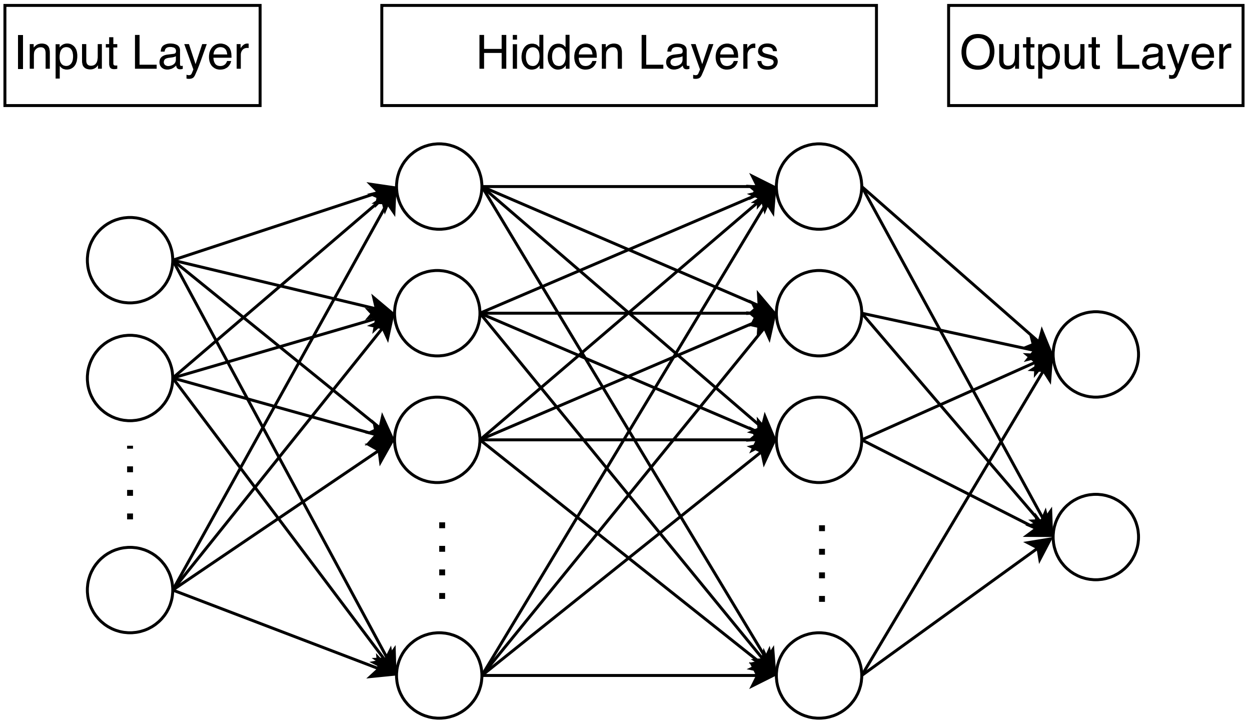


Figure 3.1: Diagram showing the input, hidden, and output layers of a typical fully connected neural network.

more output values. Figure 3.1 shows the relationship between layers in a neural network consisting of an input layer, one or more internal *hidden* layers, and an output layer. In a basic neural network, each node in a layer is connected to every node in the next layer.

Each neuron in a neural network takes inputs from a previous layer in the network or the initial inputs and then outputs a single value, $y = f(b + \sum x_i w_i)$. Figure 3.2 illustrates the calculation. The inputs to the neuron, x_i, are multiplied by weights, w_i, and summed. An optional bias, b, is also added. The sum is then passed through a function, f, known as the *activation function*. Each node in the hidden and output layers of Figure 3.1 would perform the calculation detailed in Figure 3.2 to produce its output value.

The activation function for the hidden units must be nonlinear in order for the network to learn to approximate a nonlinear function [10]. Sigmoid functions, which have an "S" shaped curve such as the logistic function, $f(x) = \frac{1}{1+e^{-x}}$, are frequently used as nonlinear activation functions in neural networks. The Rectified Linear Unit (ReLU), defined as

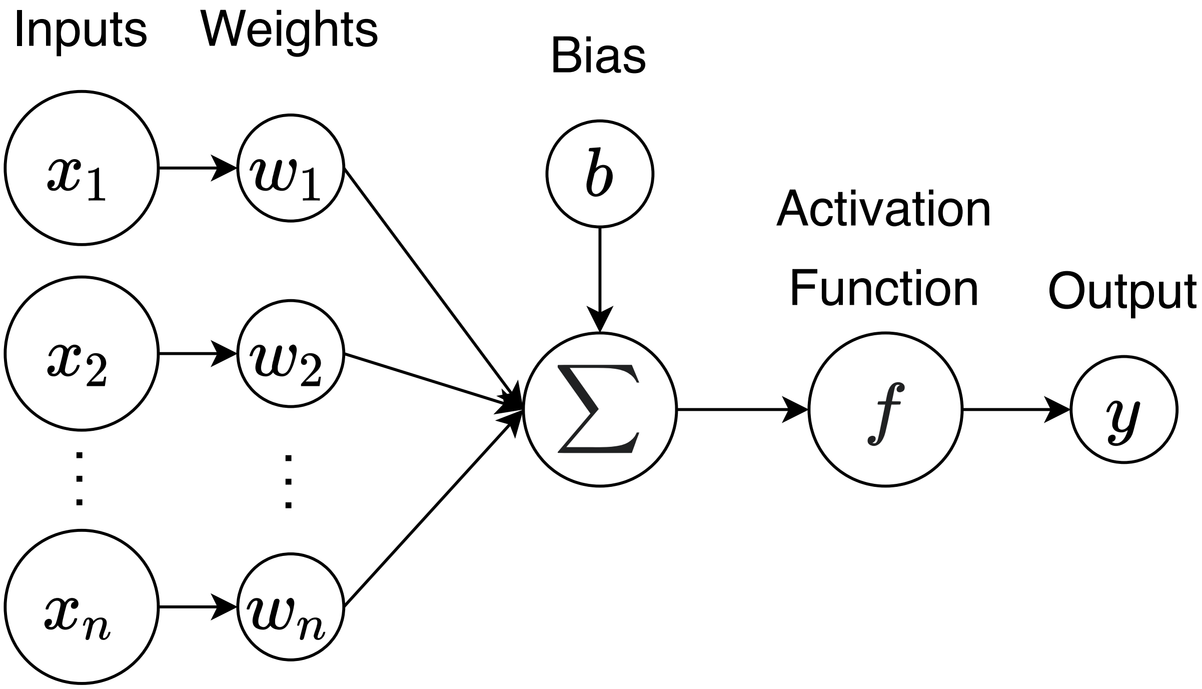


Figure 3.2: Diagram showing the calculation of an artificial neuron's output: $y = f(b + \sum x_i w_i)$.

$f(x) = \max(0, x)$, is now more commonly used, since it is easier to compute and has good performance [10]. Several universal approximation theorems have been proved, showing that neural networks can represent functions to arbitrary degrees of accuracy by changing the structure of the network, as long as certain activation functions, such as ReLUs, are used for the hidden units [10].

All the weights and biases in the network make up its parameters, denoted θ, which are modified as the network is trained. Those parameters are randomly initialized. In supervised learning, the network is trained by comparing the output of the network to a ground truth value which comes from labelled outputs, for example in a classification problem. A scalar error or cost, $E(\theta)$, is calculated using the network output and labelled output, and the network parameters are updated to reduce the error. Similarly, in deep RL an error is calculated with respect to an estimated optimal value. Common error or loss functions are mean squared error (MSE) and cross-entropy [10]. In unsupervised learning there is no labelled training data or ground truth to compare to. One example of unsupervised learning are *autoencoders* [18]. In an autoencoder the goal is to learn an efficient compressed representation of the input by training the network to output the same values as the input. The compressed representation is some inner layer of the network which is smaller than the input size and can be used in another application.

Training of neural networks is usually done with a version of stochastic gradient descent (SGD), in which we update the values of parameters in the direction that minimizes the error produced by a training step. To determine the direction and magnitude of change of each parameter, a technique called back-propagation, or *backprop*, is used to calculate the gradient of the error with respect to each parameter of the network: $\nabla_\theta E(\theta)$. This is done after a forward pass of the network produces the error value. Starting from the last layer in the network, the gradient with respect to each parameter is calculated going back layer by layer through repeated use of the chain rule. Consequently, every operation in the network must be differentiable in order for any parameters which come before it to be updated in training. Once the gradient with respect to each parameter is computed, each parameter is updated in the direction that minimizes the error. The update amount is multiplied by a decreasing learning rate, which leads to larger steps early in training and smaller steps as parameters near some minimum. This method does not guarantee that a global minimum error rate will be found, as the network can get stuck in local minima. There are numerous enhancements to SGD, including *Adam* [20] (derived from “adaptive moment estimation”), which we use in this thesis. Adam maintains an individual learning rate for each parameter in the network which is adjusted based on the frequency and magnitude of updates to the parameter. Most deep learning is done by training on batches of multiple examples in a single forward and backward pass of the network (referred to as *minibatch* training). This takes advantage of parallelization if training on a GPU, and serves as a kind of regularization since a single high error that might shift the network too much in one direction would instead be combined with the errors generated by many other training examples [10].

3.1.2 Convolutional Neural Networks

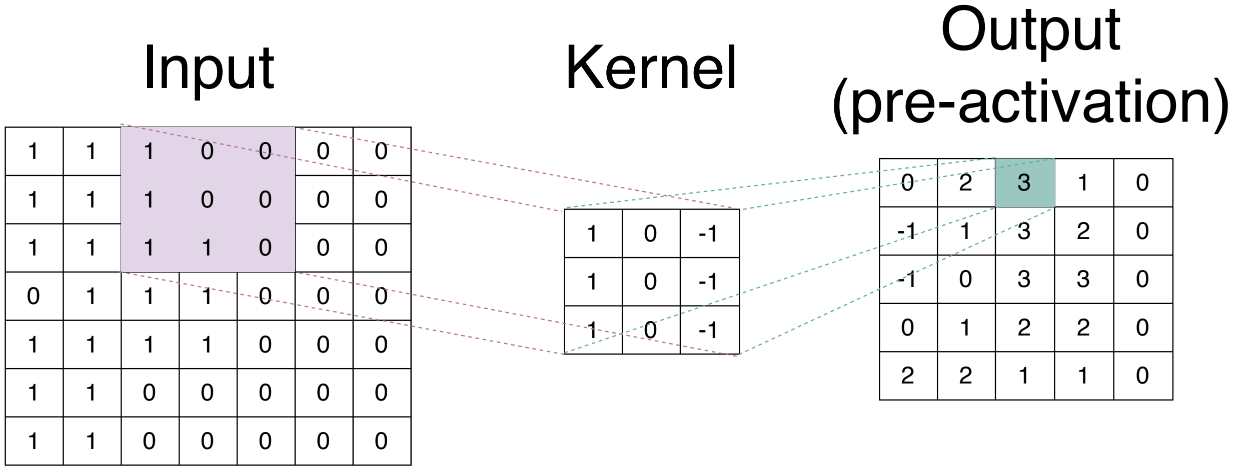
CNNs refer to neural networks partially or entirely made up of convolutional layers, which are designed to learn to detect features in data that has a spatial relationship. Recognition of 2D images is a common example, but the input data can be of any dimension. A convolutional layer consists of one or more *kernels* or *filters*, arrays of weights of the same dimension as the input, and optionally a bias for each kernel. The kernel size is smaller than the input size in the first $n-1$ dimensions for an n-dimensional input and equal to the input size in the last dimension. 2D convolution means that the input and kernel have 3 dimensions, but the kernel is moved across 2 dimensions. To obtain the output we slide the kernel along every position in the first $n-1$ dimensions and take the sum of the products of each element in the kernel multiplied with its corresponding element in the input as the output value. Ignoring the third dimension, the output is given by

$$Y(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n),$$

where the input is I and the kernel is K, with $K(0,0)$ being the centre of the kernel. This process is illustrated in Figure 3.3, though the third dimension is 1 in this example and is ignored. This operation is actually a *cross-correlation* and not a convolution (in which the kernel would be flipped), but in deep learning libraries the operation is typically called convolution and implemented as cross-correlation [10].

In Figure 3.3 a 3x3x1 kernel is applied to a 7x7x1 input, and only *valid* positions of the kernel are used, giving a smaller sized output. *Padding* (zeros, usually) on the outside edges of the input can be used to maintain the same dimensions in the output. In some cases a *stride* is used, meaning that some possible positions of the kernel are skipped, further reducing the size of the output.

For CNNs processing 2D images, the third dimension would be colour *channels* (e.g. RGB), and this convention extends to the last dimension being called the channel dimension



1	1	1	0	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0
0	1	1	1	0	0	0
1	1	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	0	0	0	0

1	0	-1
1	0	-1
1	0	-1

0	2	3	1	0
-1	1	3	2	0
-1	0	3	3	0
0	1	2	2	0
2	2	1	1	0

Figure 3.3: Diagram showing calculation of a convolutional layer output before activation function. A 3x3x1 kernel is passed over a 7x7x1 input without padding to create a 5x5x1 output.

when working with other kinds of data as well. Additionally, a convolutional layer consists of multiple filters or kernels applied to the same input, and the output of each filter is stacked along the channel dimension. Thus for 2D convolutions the input and output are in 3 dimensions. The final output of a series of convolutional layers is usually passed to one or more fully connected layers after being flattened to 1 dimension.

Using CNNs is beneficial for input such as images because a small number of weights are shared for a much larger input, and features will be detected in any location of the input [10]. To detect some feature in an image with a fully connected layer would require many more weights than a CNN, and the same patterns would need to be learned in multiple parts of the layer. By moving the kernel across the input, it can detect the same feature in multiple parts of the input.

Pooling layers reduce the size of an input by combining neighbourhoods of the input into a single output value. Several types of pooling are often used along with convolutional layers (often alternating between several convolutional layers and a pooling layer). *Average* and *max* pooling layers, for example, use filters that take the average or maximum value, respectively, of the inputs in an area and output that value in a similar way to a convolutional layer kernel. 2D pooling layers used with 2D convolutional layers (which actually have 3D kernels) will be applied with a 2D filter to all channels and the output will have the same channel dimension as the input. Stride is often used with pooling layers to reduce the size of the output. For example a 3x3 pooling layer with stride 3 will consider each pixel of its input once as it is moved over it.

3.2 Reinforcement Learning

RL is a type of ML in which a system learns to take actions in an environment to maximize expected future reward. In this section we describe important concepts in RL and provide the foundation for the algorithm we use in this thesis, DQN. DQN is a deep learning version of Q-learning, which is a temporal difference (TD) RL method, so we focus on the concepts necessary for understanding TD methods before introducing Q-learning. The concepts covered here are presented in complete detail in the book *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew Barto [42]. We usually follow the notations used in that book, with occasional deviations for simplicity.

3.2.1 Important Concepts in Reinforcement Learning

Figure 3.4 shows the relationship between an RL agent and an environment. At time step t the agent observes state s_t and reward r_t from the environment and takes action a_t. Applying the action a_t to the environment advances it to a new state s_{t+1}, and the state and a new reward r_{t+1} are observed by the agent. The reward r_{t+1} tells the agent how desirable it is to be in state s_{t+1}. Equivalently, we sometimes refer to a state s and the action taken at s, a, along with the next state and next action, s' and a', and reward at s', r.

An RL algorithm can involve estimating a value function, $v_\pi(s)$, which gives the amount of future rewards (the return) expected from a given state s when following a policy π, or $Q_\pi(s, a)$, the value of taking action a in state s when following policy π. A policy function, $\pi(s)$, tells the agent which actions to take in a particular state. The estimations of the

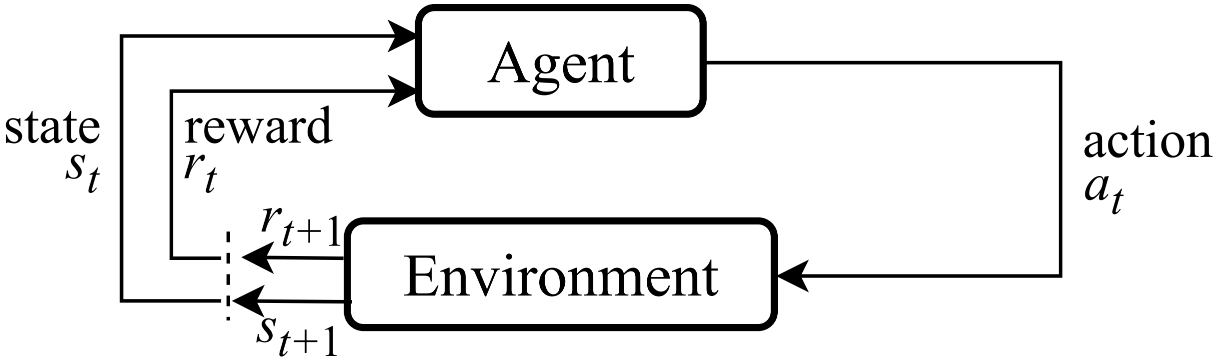


Figure 3.4: Diagram showing the flow in RL of state and reward information from the environment to the agent, and actions from agent to environment.

value and optimal policy functions get better over time through repeated experiences in the environment. If a state/action pair (s, a) leads to a positive (negative) reward r, the estimation of its value or the likelihood of taking that action in the same state is increased (decreased). Policies can be stochastic or deterministic.

RL differs from supervised learning, where the correct actions are labelled, and from unsupervised learning, which can be used to learn patterns in data. RL does have similarities to supervised learning in that the reward function is determined by a human designer and guides the learning. RL algorithms can be used with or without a model of the environment (which would give knowledge of the reward and next-state obtained from taking an action), and updates to the policy during training can be made using actions taken with the current policy (on-policy) or not (off-policy).

3.2.1.1 Exploration vs. Exploitation

One of the basic problems in RL is known as *exploration vs. exploitation.* Ultimately we want an agent to take the best actions, the ones which lead to the highest reward. Taking what is believed to be the best action at any time, the *greedy* action, is exploitation. The agent needs to explore some of the time by taking actions which it thinks are not the best in order to get a better estimate of how good those action are. For a single state environment this is known as the *k-armed bandit problem*, where there are k actions to choose from at each time step t, and taking action a_t gives reward r_t [42]. The reward for taking an action is sampled from a probability distribution, and the expected reward or value of taking action a is given by

$$Q^*(a) = \mathbb{E}\left[r_t | a_t = a\right].$$

The estimate of the value of taking action a at time t is $Q_t(a)$. One way to handle this problem, called ϵ-greedy, is to take the greedy action most of the time and to take a random action with a small probability of ϵ. The greedy action is given by

$$a_t = \underset{a}{\operatorname{argmax}} Q_t(a).$$

The incremental update rule for the value of taking an action in a bandit problem is

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N(a)}\left[r_{t+1} + Q_t(a)\right], \tag{3.1}$$

where $N(a)$ is the number of times action a has been chosen. If the problem is nonstationary, meaning that the distribution of rewards changes over time, it makes sense to use a weighted average and to assign higher weights to more recent rewards [42]. In that case the update rule is given by

$$Q_{t+1}(a) = Q_t(a) + \alpha\left[r_{t+1} + Q_t(a)\right],$$

where $\alpha \in (0, 1]$ is a constant step-size parameter. Alternatively, on-policy algorithms with stochastic policies can maintain a nonzero probability of taking all actions and explore naturally just by following the policy.

3.2.1.2 Finite Markov Decision Processes

If a finite environment has the property that the next state, s', is only dependent on the current state s and action a, not on previous states and actions, it is said to be a finite Markov decision process (MDP), or to have the *Markov property.* In other words, the current state contains all the information necessary to know the value of that state, i.e. the expected future rewards. In practice many RL environments are not MDPs, but the property is assumed for many proofs about RL algorithms [42].

The expected future rewards from a state s_t (the state at time t) is formally called the *expected return*, given by

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} \cdots + r_T,$$

where T is the final time step. The final step would be the last step in an episode if the environment is not a continuous task and plays out in separate episodes. Similar to the step-size parameter in Equation 3.1, we can discount future rewards so that rewards that come sooner are given higher weight:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma \in (0, 1)$ is the *discount rate*. This leads to an important relationship relating the return in time step t to the return in time step $t + 1$:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \\ &= r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \cdots \right) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned}$$

The state-value function $v_\pi(s)$ can now be defined as the expected return when following policy π from state s:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right].$$

Similarly, the action-value function $Q_\pi(s, a)$ can be defined as:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right].$$

The value of a state and its successor states are related by the Bellman equation, given by:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid s_t = s \right] \\
&= \mathbb{E}_\pi \left[r_{t+1} + \gamma G_{t+1} \mid s_t = s \right] \\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r P\left(s', r \mid s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid s_{t+1} = s' \right] \right] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} P\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right) \right],
\end{aligned}$$

where s' is the state that occurs after action a is taken in state s, and $P(s', r \mid s, a)$ is the probability of transitioning to s' and receiving reward r if the action a is taken in state s. This relationship shows how the value of a state can be updated based on a previous estimate of the value of the next state and the reward obtained when moving to the next state [42].

The Bellman equation leads to several ways to solve RL problems. One uses dynamic programming to iteratively improve estimations of $v(s)$ for all states $s \in \mathcal{S}$, the set of all states, by using the current estimates of state-values to *bootstrap* the updates to other state-values. This is only possible when working in an environment for which there is a model, and importantly for which the number of possible states is small enough to make iterating over all of them multiple times feasible. Algorithm 1 describes one such method, *value iteration* [42]. Value iteration can find the optimal action for each state to an arbitrary level of accuracy, determined by the parameter θ.

When there is no model of the environment, RL algorithms must learn from direct experiences in the environment. *Monte Carlo* methods work by taking actions and recording rewards until reaching a terminal state, i.e. until an episode ends, and do not depend on knowing what the next state will be after taking an action (that is, they don't depend on having a model of the environment) [42]. Upon reaching a terminal state, the return, G_t, for each state, s_t, visited is calculated, and either state-values or action-values can then be updated. This process repeats for an arbitrary amount of time. Monte Carlo methods can be on-policy, in which the policy used to generate the episodes is the one being updated

Algorithm 1 Value iteration dynamic programming

```
Input: small accuracy threshold $\theta < 0$
Initialize $v(s) = 0 \forall s \in \mathcal{S}$
$\Delta \leftarrow 0$
while $\Delta < \theta$ do
    $\Delta \leftarrow 0$
    for each state $s$ do
        $v \leftarrow v(s)$
        $v(s) \leftarrow \max_a \sum_{s',r} P(s', r \mid s, a) [r + \gamma v(s')]$
        $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
Output deterministic policy $\pi \approx \pi^*$ ($\pi^*$ is the optimal policy):
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} P(s', r \mid s, a) [r + \gamma v(s')]$
```

(ϵ-greedy, for example), or off-policy, as long as the policy used to generate episodes has a nonzero probability of selecting every action in a state.

3.2.2 Tabular Temporal Difference Methods

TD RL methods combine elements of dynamic programming methods and Monte Carlo methods. They update state-value or action-value estimates one step at a time, using the estimates of one or more other states' values, and they also use experiences from taking actions in the environment, so they don't rely on having access to a model of the environment. The update rule for a one-step state-value update, known as TD(0), is

$$v(s_t) \leftarrow v(s_t) + \alpha \left[r_{t+1} + \gamma v(s_{t+1}) - v(s_t) \right].$$

$r_{t+1} + \gamma v(s_{t+1})$ is known as the *TD target*, and

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$$

is the *TD error*. This update is one-step because it uses the value of the next state and the reward at the next state to make the update. In general, TD methods can be n-step, where

Algorithm 2 Sarsa on-policy temporal difference reinforcement learning

```
Input: step size $\alpha \in (0, 1]$, discount $\gamma$, small exploration parameter $\epsilon > 0$
Initialize $Q(s, a) = 0 \; \forall s \in \mathcal{S}, a \in \mathcal{A}$
for each episode do
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    while $s$ is not terminal do
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$
```

the TD target would be $r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n v(s_{t+n})$.

Two popular TD methods for estimating action-values are *Sarsa* and *Q-learning* [42]. Sarsa is an on-policy method, while Q-learning is an off-policy method. The algorithms presented here (as with the dynamic programming and Monte Carlo methods presented in Section 3.2.1.2) are *tabular* because they only apply to environments where we can store and update action-values for *every* state within practical memory and computation limits.

3.2.2.1 Sarsa

Sarsa makes an update to $Q_\pi(s, a)$ after each step, using the reward and action-value of the next state to update the action-value of the last state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right].$$

If s_{t+1} is a terminal state then $Q(s_{t+1}, a_{t+1})$ is zero for the update. The state transition elements used, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, give the algorithm the name Sarsa. Sarsa converges to the optimal policy over an infinite number of steps as long as 1) each state-action pair has a nonzero probability of being explored throughout learning, and 2) the exploration policy converges to the greedy policy over the learning steps [42]. Algorithm 2 shows the full Sarsa procedure. $\mathcal{A}$ in the algorithm is the set of all actions.

Algorithm 3 Q-learning off-policy temporal difference reinforcement learning

```
Input: step size $\alpha \in (0, 1]$, discount $\gamma$, small exploration parameter $\epsilon > 0$
Initialize $Q(s, a) = 0 \; \forall s \in S, a \in A$
for each episode do
    Initialize $s$
    while $s$ is not terminal do
        Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
        $s \leftarrow s'$
```

3.2.2.2 Q-Learning

Q-learning is the off-policy counterpart to Sarsa. It updates action-values by using the greedy action at the next state, rather than the actual next action chosen by the policy. Because of this difference, Q-learning directly learns the optimal policy, but does not necessarily perform as well during training. Sarsa, by contrast, takes its exploration into account when making updates, so it learns the policy which is optimal given the exploration in its policy. The update rule for Q-learning is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

For Q-learning to converge to the optimal policy, the only requirement is that every state-action pair continues to be updated throughout learning (i.e. ϵ does not need to decay in ϵ-greedy exploration) [42]. The full Q-learning algorithm is presented in Algorithm 3.

The maximization in the target calculation of Q-learning can lead to an overestimation of action-values, called *maximization bias* [42]. If the maximal action at the next state, s', has an overestimated value, that overestimation propagates back to the state s during learning. A solution to reduce this effect in general is called *double learning*, or *Double Q-learning* when applied to Q-learning. Instead of learning a single estimate of action-values, $Q(s, a)$, we learn two estimates simultaneously, $Q_1(s, a)$, and $Q_2(s, a)$. When choosing the target action-value in Double Q-learning, we choose the action with the maximum value from one estimate, and use the value associated with that action in the other estimate. The choice of which estimate is used for which part of the update is random in each update. The update rule is given by

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha \left[r_{t+1} + \gamma\, Q_2(s_{t+1}, \underset{a}{\text{argmax}} Q_1(s_{t+1}, a)) - Q_1(s_t, a_t) \right],$$

where Q_1 and Q_2 are switched for half of the updates.

3.2.3 Deep Q-Network (DQN) and Enhancements

DQN is a deep learning extension of Q-learning, introduced in two works by Mnih et al. in 2013 and 2015 for learning to play classic Atari games in the ALE environment [25, 26]. DQN uses a neural network to approximate $Q(s, a)$, making it suitable for environments with large state and action spaces where it is infeasible to store and learn action-values for every state-value combination. In this section we present DQN as originally implemented for the ALE. We make use of three common enhancements to DQN which we also describe in this section: double DQN, prioritized experience replay, and a dueling network architecture.

3.2.3.1 Original DQN

DQN makes several changes from standard Q-learning which help avoid instabilities (difficulty converging) that occur when using a nonlinear function approximator such as a neural network for the action-value function [26]. Sequential experiences are likely to have correlated states, rewards, and TD errors, which can lead to large changes in the network weights too far in one direction. DQN uses *experience replay*, which stores transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ for non-terminal s_t in replay memory M to be sampled randomly for training. The experience replay holds the last N transitions, and every K steps k transitions are sampled for a minibatch of training, removing the correlation between transitions as the network is trained.

DQN also uses two networks representing the action-value function, a primary network,

Q, with parameters θ, and a second target network, $\hat{Q}$, with parameters θ^-. Q and $\hat{Q}$ are identical in structure, but $\hat{Q}$ is kept as a stale version of Q, with θ^- updated to θ every C steps. $\hat{Q}$ is used to generate the TD targets $y = r + \gamma \underset{a'}{\max} \hat{Q}(s', a'; \theta^-)$. Adding this delay to updates of the target network was found to improve the stability of the algorithm by making large oscillations in the policy less likely [26].

The last technique used to improve stability in DQN is to clip the gradient of the error to the interval $[-1, 1]$, which is equivalent to using MSE as an error function for TD error between -1 and 1, and the absolute value error function outside that interval [26]. The loss function is given by:

$$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| < 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases}, \tag{3.2}$$

where $\delta = y - Q(s, a; \theta)$ is the TD error. This function is known as the *Huber* loss function [17]. Clipping the gradient of the loss in this way prevents large errors from resulting in large changes to the network in a single training update.

Algorithm 4 shows the complete DQN procedure with an ϵ-greedy policy. In the initial ALE environment work [26], ϵ-greedy is used as the policy, with ϵ linearly annealed from 1 to 0.1 over the course of the first 1M steps of training. The experience replay memory had a size of 1M and was filed with an initial 50,000 transitions following a uniform random policy before any learning started. SGD updates were performed every 4 steps with a minibatch size of 32, using the RMSProp (Root Mean Square Propagation) [14] optimizing algorithm, which is an SGD method that maintains a separate learning rate for each parameter.

Algorithm 4 samples a minibatch of transitions on step 17 and calculates the loss for each transition on step 20. Note that these calculations happen in parallel for all transitions in a minibatch, rather than in a loop. The actual final scalar loss that is minimized is the mean of the losses from all transitions in the minibatch.

Algorithm 4 DQN with experience replay

```
Input: discount $\gamma$, initial $\epsilon$ and $\epsilon$ annealing schedule, minibatch size $k$, batch update frequency $K$, target update frequency $C$, max steps $T$, memory size $N$, and $M_{min} \leq N$
Initialize replay memory $M \leftarrow \emptyset$
Initialize $Q$ with random parameters $\theta$, $\hat{Q}$ with parameters $\theta^{-} = \theta$
Initialize environment with state $s$
for $t = 1$ to $T$ do
    if rand() $< \epsilon$ or $t \leq M_{min}$ then
        Select random action $a$
    else
        $a \leftarrow \underset{a_t}{\text{argmax}} Q(s, a_t; \theta)$
    Take action $a$ in environment, observe $s', r$
    Store transition $(s, a, r, s')$ in $M$
    $s \leftarrow s'$
    Anneal $\epsilon$ according to schedule
    if $t \equiv 0 \mod C$ then
        Update weights $\theta^{-} \leftarrow \theta$
    if $t \equiv 0 \mod K$ and $t > M_{min}$ then
        Sample random minibatch of $k$ transitions $(s_j, a_j, r_j, s'_j)$ from $M$
        $y_j = \begin{cases} r_j & \text{if } s'_j \text{ is terminal} \\ r_j + \gamma \underset{a'}{\max} \hat{Q}(s'_j, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Compute TD error $\delta_j = y_j - Q(s_j, a_j; \theta)$
        Update $\theta$ with gradient descent step minimizing $L(\delta_j)$
```

3.2.3.2 Double DQN

As explained in Section 3.2.2.2, tabular Q-learning is known to overestimate action-values due to the max operation in the selection of the target action. The same effect has been shown to exist with DQN [48]. As with tabular Double q-learning, in Double DQN a second estimation of $Q(s, a)$ is used to separate the choice of target action and the value of the target action. Since DQN already has two copies of the Q-network, one of which is an out-of-date copy of the other, in Double DQN we use the primary network to choose the target action, and the target network to estimate it's value. The Double DQN target is given by:

$$y = r + \gamma \hat{Q}(s', \underset{a'}{\text{argmax}} Q(s', a'; \theta); \theta^{-}).$$

This change is meant to give the benefit of double learning with the smallest possible increase in memory use and computation, and was found to improve action-value accuracy and policy performance [48]. Note that it does require minibatch training to require two forward passes of the neural network instead of one. In the first pass, the actions with the maximum action-value for each transition's next-state are output from Q. In the second pass, $Q(s, a; \theta)$, the predicted action-value of the action taken in the transition, is generated with Q, and $\hat{Q}(s', a'; \theta^{-})$ is generated with $\hat{Q}$ (where a' was found in the first pass), giving all the values needed to calculate the TD error. Algorithm 5 shows the full Double DQN algorithm with prioritized experience replay, which is explained in Section 3.2.3.3

3.2.3.3 Prioritized Experience Replay

Prioritized experience replay aims to increase the frequency of *important* transitions being sampled for minibatch training [34]. It does that by assigning a priority proportional to the magnitude of the TD error generated by a transition the last time it was sampled (a measure of how "surprising or unexpected" a transition is), and using that priority when sampling from the replay memory [34]. The practice of keeping only the last N transitions is unchanged, and all transitions maintain a nonzero probability of being sampled. Priorities are stored in a *sum-tree* data structure which allows efficient updates and sampling. Algorithm 5 shows the Double DQN algorithm with prioritized experience replay.

When a transition is first stored in the replay memory (see Step 11) its priority is set to be equal to the maximum priority of any transition saved up to that point (with a default initial maximum priority), so that it will likely be sampled at least once. Because the priority of transitions are only updated when they are sampled, there can be transitions that would be useful to sample at some step in training but have a low priority due to not being sampled recently. To increase the diversity of samples, the probability of sampling a transition is not

Algorithm 5 Double DQN with prioritized experience replay

```
Input: discount $\gamma$, initial $\epsilon$ and $\epsilon$ annealing schedule, minibatch size $k$, batch update frequency $K$, target update frequency $C$, max steps $T$, memory size $N$, $M_{min} \leq N$, prioritized experience replay exponents $\alpha$ and $\beta$, and $\beta$ annealing schedule
Initialize replay memory $M \leftarrow \emptyset$
Initialize $Q$ with random parameters $\theta$, $\hat{Q}$ with parameters $\theta^{-} = \theta$
Initialize environment with state $s$
for $t = 1$ to $T$ do
    if rand() $< \epsilon$ or $t \leq M_{min}$ then
        Select random action $a$
    else
        $a \leftarrow \underset{a_t}{\operatorname{argmax}} Q(s, a_t; \theta)$
    Take action $a$ in environment, observe $s', r$
    Store transition $(s, a, r, s')$ in $M$ with maximal priority $p_t = \max_i(p_i)$
    $s \leftarrow s'$
    Anneal $\epsilon$, increase $\beta$ according to schedules
    if $t \equiv 0 \mod C$ then
        Update weights $\theta^{-} \leftarrow \theta$
    if $t \equiv 0 \mod K$ and $t > M_{min}$ then
        Sample minibatch of $k$ transitions $(s_j, a_j, r_j, s'_j)$ from $M$ with $P(j) = p_j^\alpha / \sum_i p_i^\alpha$
        Compute importance sampling weights $w_j = (N \cdot P(j))^{-\beta} / \max_i(w_i)$
        $y_j = \begin{cases} r_j & \text{if } s'_j \text{ is terminal} \\ r_j + \gamma \hat{Q}(s'_j, \underset{a'}{\operatorname{argmax}} Q(s'_j, a'; \theta); \theta^{-}) & \text{otherwise} \end{cases}$
        Compute TD error $\delta_j = y_j - Q(s_j, a_j; \theta)$
        Update transition priority $p_j \leftarrow |\delta_j|$
        Update $\theta$ with gradient descent step minimizing $L(w_j \delta_j)$
```

dependent only on its priority. The probability of sampling transition i is given by

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where p_i is the priority of transition i. α controls the amount of weight given to prioritization (higher α) or to random sampling ($\alpha = 0$). Because of the p_i^α terms in the probability calculation, it is actually p_i^α which is stored in the sum-tree, rather than p_i.

Prioritized experience replay also uses *importance sampling* (IS) weights, which reduce the magnitude of the TD error for transitions when performing training updates. IS weights are used to compensate for bias introduced by sampling transitions in a non-uniform random way, and reduce the possible over-corrective effect of sampling the same high error transitions many times [34]. The weight of transition i is given by

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta},$$

where $\beta \leq 1$ determines the amount of compensation. The weight is multiplied by the TD error on step 22. The authors use $\alpha = 0.6$ and linearly increase β from 0.4 to 1 over the course of training when using Double DQN, reasoning that removing bias caused by the prioritized sampling is more important as the policy converges near the end of training [34].

3.2.3.4 Dueling Network Architecture

A dueling network architecture aims to allow a Q-network to independently learn both the value of being in a state and the *advantage* of taking a particular action in that state. There is no change to the RL algorithm, only to the structure of the network, and in fact dueling architectures can be used with many RL algorithms. This technique was shown to produce better performance when used with DQN in the ALE environment than original DQN, and particularly improves performance in situations (states) where there are many actions which have the same value [52].

The change in network design is to split the network into two streams before the action-value output. For the original DQN work [52], the authors split the streams after an initial CNN portion. The value and action advantage streams each consist of fully connected layers. The value stream ends with a layer of width 1, i.e. a scalar, and the advantage stream ends with a layer with width the same as the number of actions. Then the two streams are combined to output the action-values of each action.

The advantage function is defined as:

$$A_\pi(s, a) = Q_\pi(s, a) - v_\pi(s). \tag{3.3}$$

By subtracting the value of a state from the action-value of taking an action in that state, we get the relative benefit of taking the action in that state. Rearranging Equation 3.3 gives one way to combine the advantage and value streams to produce the action value output: $Q(s, a) = v(s) + A(s, a)$. However, under the optimal policy the action-value of the best action, a^*, in a given state and the state-value of that state are the same, meaning that $A(s, a^*) = 0$. To ensure there is zero advantage at the chosen action, the authors propose

$$Q(s, a) = v(s) + \left(A(s, a) - \max_{a'}(A(s, a'))\right).$$

In practice, they found that subtracting by the average advantage, instead, resulted in more stable performance:

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

Chapter 4

Methodology

In this chapter we present our methods for training RL players using the SC2LE. In Section 4.1 we describe the state and action representations of the SC2LE, focusing on the challenges in training an RL player to use them. The microRTS environment can be approached similarly with respect to action representation, and in Section 4.2 we describe our work in adapting our RL methods to that environment. In Section 4.3 we describe our implementation of the DQN RL algorithm modified to work with component actions in detail. In Section 4.4 we describe the custom StarCraft II scenarios we created and use in our experiments.

4.1 StarCraft II Learning Environment

The SC2LE includes both C++ and Python APIs for playing the full game of SC2. The C++ API is similar to the BWAPI for StarCraft: BroodWar, and allows an AI player to issue an arbitrary number of commands directly to units. For example, every unit belonging to an AI player can receive an independent order on every frame. Precise information on every visible unit is available in the gamestate. The Python interface (PySC2), designed for ML/RL research, more closely emulates how a human plays the game by presenting similar information that human would have access to visually, and by only allowing actions similar to those a human could make on a single frame. Figure 1.1 shows the human StarCraft II interface, while Figure 2.1 shows a visualization of all the PySC2 feature maps.

In PySC2, every n frames an RL player receives a state observation and then specifies an action to take, where n is adjustable. The game normally runs at 24 frames per second, and all our experiments have the RL player acting every 8 frames, as in previous work [51]. At this speed the RL player acts at up to 180 APM, a similar rate as a human [49], and the gamestate can change meaningfully between states. The APM is up to 180 because the RL player has the option to perform no action. Any player controlled with PySC2 will still have its units automated to an extent by built-in AI, as with a human player. For example, if units are standing idly and an enemy enters within range, they will attack that enemy.

4.1.1 PySC2 Observations

PySC2 observations consist of a number of 2D feature maps conveying categorical and scalar information from the main game map and minimap, detailed below, as well as a 1D vector of other information that does not have a spatial component. Feature maps are easily used with CNNs in deep learning, as they represent meaningful data about the game state. Conversely, PySC2 also includes a full RGB screen observation mode, where the input is approximately the RGB channels of the image that a human player would see on screen (the normal game view of the map is skewed slightly whereas PySC2 presents a top-down view of the map). This type of state observation would be much harder to learn from, since some equivalent of the feature map information would need to be learned from the RGB data by the neural network. That is, at some intermediate layer of the neural network, features similar to the provided feature maps would have to be learned from the RGB data to then make generalizations about those features. By using the feature layers provided directly by PySC2, the neural network layers closest to the input can learn generalizations about the features. No other works have been published using the RGB mode in PySC2 and we did not use it in our research.

Feature map information included in the state observation includes: various unit properties such as health and shield values, terrain data, and player (unit/building owner) information. The 1D vector includes player resource amounts, army size, score, and other information not contained in the map. All categorical and scalar observation data are given as integers. The resolutions of the feature maps for both the main map and minimap are configurable in PySC2. Unit positions on the map are real-valued, so there is no natural resolution for the feature maps. We use a resolution of (84,84) in all our experiments, as used in the paper presenting PySC2 [51]. Different units may be represented by different numbers of cells in the 2D feature maps based on the size and position of the unit on the screen and the map resolution.

The observation also includes a list of valid actions (detailed in the next subsection) for the current frame which we use to mask out illegal actions. This is equivalent to information a human player could glean from the visible game interface. For instance, if a transport unit is currently selected and it is currently transporting units, an "unload all" button would be available in the human interface. In PySC2 the corresponding function for "unload all" would be in the valid actions list.

In our experiments we use a subset of the 27 main map spatial features relevant to the combat scenarios we use. In our scenarios it is useful to the player to know where the units are, which units are currently selected, which units belong to the player and opponent, and about the health and other dynamic properties of the units. We haven't used scenarios that include buildings or differences in terrain, so we don't use features indicating where buildings can be placed, for example, or what the height of the terrain is. We also don't use a feature indicating unit type, since each of our scenarios feature only one unit type at a time. The features we use are:

player_relative - categorical feature describing if units are controlled by the AI player (self) or enemy (and some other categories we don't use, such as neutral units)

selected - Boolean feature showing which units are currently selected (i.e. if the player can give them commands)

unit_hit_points - scalar feature giving remaining health of units, which we convert to three or four categories, depending on the maximum health of the units being trained

unit_shields - scalar feature giving the remaining shield value of units, which we convert to four categories

buffs - categorical feature indicating if a unit has an active "buff" or benefit granted by using a special ability

4.1.2 PySC2 Actions

Actions in PySC2 are conceptually similar to how a human player interacts with the game using a keyboard and mouse. There are three ways a human player controls the game:

1. Selecting units and buildings, by left clicking on one to select it, clicking and dragging a rectangle shape to select all units in the rectangle (units can also be added to a selection by holding a key), or by hitting keys to save or recall a selection of a group of units (called a "control group");

2. Right clicking on the map or minimap to perform a contextual action with the selected units, such as attacking an enemy if clicking on an enemy, or moving to a location if clicking on unoccupied space; and

3. Performing a specific action or ability with the selected units by either clicking a corresponding icon in the unit menu or by pressing a specific "hot-key" (e.g. "A" for "attack"), and possibly then left-clicking on a target for the ability if applicable.

PySC2 actions consist of an action *function* selection and 0 or more arguments to the function (the number of arguments depends on the function). The function and all argument inputs are integers corresponding to integer IDs specified in PySC2. There are over 500 action functions, consisting of "no op" (take no action); "move camera" (move the camera view to a point referenced on the minimap); multiple ways to select units, move units, and attack; and hundreds of abilities unique to individual units or groups of units. Selecting units can be accomplished in several ways, including: 1) select a unit whose shape overlaps with a specific point on the map, 2) select all the units contained in a rectangle specified by two opposing corner points (as a human does with a mouse), 3) select all available army units, and 4) recall a selection of units previously saved to a control group. Movement can also be a direct move to a point, movement while stopping to attack enemies in range, or a patrol action causing a unit to move back and forth between points. Many action functions have more generic versions that will do the same thing when used with different units. For example there is a generic "burrow" function as well as specific burrow functions for many Zerg units, which causes them to hide underground. The generic function could be used in all cases to achieve the same effect.

Action functions have argument types that are reused by many functions, so the list of all argument types is small compared to the number of functions. Functions that require a screen or minimap location as an argument take a coordinate tuple as an argument. Commands to a unit in StarCraft II can be queued, so that the unit will perform the commands in order (a unit can be instructed to move to a series of points in order, for example). Similarly, many action functions in PySC2 can also be queued, which is a binary argument to those functions. Many actions that a human would consider to be one action, such as moving the mouse to a specific point on the screen and right-clicking, are translated as a single action in PySC2. Choosing to build a building (pressing a key or clicking an icon) and then placing that building on the map (clicking on the map) might be considered to be two actions by a human, but it is just one action in PySC2.

We use a small subset of the action functions in PySC2 which are sufficient for the combat scenarios in our experiments. We want the smallest possible subset of actions for efficient combat, because more actions makes the learning task harder. The neural network needs to choose from among the action functions by learning an approximation of the value or advantage of taking one action over another in a given state. With more action choices, it takes more experimentation, i.e. more training time, to learn those values. Therefore we exclude action functions that are repetitive or irrelevant to our combat scenarios. For example, we don't use the action function that allows the player to select a single point, but we do allow selecting a rectangle, since selecting a small rectangle would be functionally equivalent. The action functions we use and their parameters are:

no_op - do nothing

select_rect - select units in a rectangular region

- screen (x, y) (top-left position)
- screen2 (x, y) (bottom-right position)

select_army - select all friendly combat units

move_screen - move to a position while ignoring enemies

- screen (x, y)

attack_screen - attack unit or move towards a position and stop to attack enemies in range on the way

- screen (x, y)

4.2 microRTS

microRTS is a simple open source RTS game designed for AI research and released in 2013 [27]. It has many of the same elements as commercial RTS games, including 3 types of combat units, workers, buildings, resources, and different types of map terrain (open or wall). Maps in the game are 2D grids allowing 4 way movement for units, and ones used in its official competition range from 8x8 to 64x64 tiles. The game has complete and incomplete

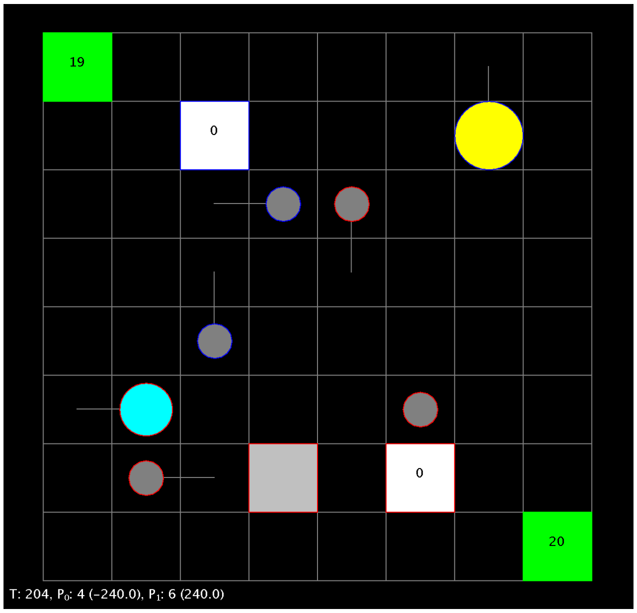


Figure 4.1: A visualization of a microRTS match on a 8x8 tile map.

information modes, where either the entire map is visible or only tiles within a certain range of friendly units. Figure 4.1 shows a visualization of a game in progress on a 8x8 map.

microRTS has been used to compare heuristic search techniques for RTS environments through an annual competition since 2017 [28]. It has also been used for evaluating neural network backed state evaluation for use in heuristic search [40]. Limited work [16] with RL has been done in microRTS.

Since microRTS is considered to be a simpler version of an RTS that is a good testbed for RTS AI research, we attempted to adapt our implementation of component-action DQN for PySC2 to microRTS early on in this research to see if it could be used to test the feasibility of the algorithm. Since maps are small this environment also has the benefit of allowing faster training. microRTS is implemented in Java and AI bots written for it can be Java classes or can be implemented in another language, communicating with the game over socket. We created an interface in Python using sockets, so that we could use much of the same code we were developing for using PySC2.

4.2.1 microRTS Action Representation

Since microRTS is played on a grid, creating a feature map state representation was straightforward and similar to the state representation in PySC2. However, microRTS has some major differences to most RTS games in it's action interface, which we believe made it difficult to adapt well to the system we are using for playing sub-sections of StarCraft II. microRTS allows bots to issue an action to each unit it controls on every frame of the game. Since our goal in StarCraft II is to have a RL player control the game like a human would, we could have limited our DQN agent playing microRTS to issuing one command to one unit with a delay before acting again, as was later used in [16]. We thought that method would not result in good performance against other microRTS bots since they would not be constrained in the same way. We could also have implemented an abstraction to allow the agent to select multiple units at once (similar to how that is done in pySC2) to make better use of the limited actions. Instead we allowed the agent to make multiple actions per frame with as much granularity as possible, while also keeping systems used in our PySC2 interface as much as possible.

microRTS allows each unit to move one tile up, left, down, or right. Combat units and workers can attack any tile within its range (1-3) by Manhattan distance, meaning distance counted with 4-way movement. Workers can harvest resources from a tile that is in range 1 and return resources to a base within range 1, as well as build buildings within range 1, while buildings can produce new units in empty tiles within range 1. Actions in microRTS also have duration, so a unit that is still completing an action cannot be issued a new one.

In microRTS, an action for a turn or frame of the game is a list of unit-actions for each friendly idle unit consists of these components:

type - one of none/move/attack_location/harvest/return/produce

parameter - one of none/up/right/down/left (used for direction of move, harvest, return, and produce)

x - x coordinate of attack (if attacking)

y - y coordinate of attack (if attacking)

unitType - type of unit/building to produce (if producing)

To make the action specification more closely resemble the actions in PySC2 we mapped that specification to our own which combines the x, y, and parameter components into a single coordinate component. The network outputs:

select - (x,y) coordinates of a unit to issue a command to

function - one of none/move/attack/harvest/return/produce

target - (x,y) coordinates for a target tile (used for all actions except "none")

unitType - type of unit/building to produce (if producing)

Unlike in our PySC2 interface where the function component action is chosen first, which determines which other components are used, the dependencies are more complex for our microRTS actions. Here legal options for the target and unitType components depend on the function component choice, and legal options for the function component depend on the select component choice. Using those four components we create individual unit-actions. To create a complete action, we output a new unit-action from the network once for each idle unit needing a command, but with the state adjusted each time to reflect which units were occupied with a previous command (action types in progress and their time to completion were included in the state representation). It was also necessary to filter each component's output based on legal moves. Filtering of legal actions was complicated and error-prone to implement in an efficient way to operate on the arrays output by the network.

Formally, the agent receives state s_t from the microRTS. Each unit-action $a_{t,0}, a_{t,1}, \ldots,$ $a_{t,n-1}$ for n idle units is generated from $s_{t,0} \equiv s_t, s_{t,1}, ..., s_{t,n-1}$, where each successive state is updated with information about the previous unit-action. A complete action a_t consisting of all the unit-actions for the turn is sent to microRTS, advancing the game until the next time a unit needs orders. Since orders have duration, new orders won't be issued on every frame of the game. When the agent receives the reward r_t and next state s_{t+1}, we can store all transitions $(s_{t,i}, a_{t,i}, r_t, s_{t+1})$ for $0 \leq i < n$ in the replay memory. Since the start states for many transitions would be very similar we experimented with both storing transitions for all n unit-actions and with randomly selecting one of those transitions to store.

4.3 Component-Action DQN

As explained in Section 3.2.2.2, Q-learning is well suited to environments where we only care about the final performance, not performance during training, such as PySC2. Because the state space for PySC2 is very large, even when using only a subset of the feature maps available, we chose to apply a neural network-based version of Q-learning, DQN [26], to this environment. Additionally, we use several enhancements to DQN which have been shown to result in better performance: the Double DQN enhancement [48], prioritized experience replay [34], and a dueling network architecture [52]. We use an ϵ-greedy policy for exploration, as in the original DQN. Noisy network exploration has been applied to DQN in Rainbow [13] with better performance when compared to ϵ-greedy exploration, but that method can only be applied to network outputs ending in dense layers. Our network is structured differently, as will be described in Section 4.3.3. To address the needs of the PySC2 action interface we divide the output of the network into what we call *action components*. We also tested different methods for calculating loss while using action components.

Our version of DQN is custom-implemented in Python and uses TensorFlow [1] to build and update the neural network. At first, we investigated using existing RL libraries such as Tensorforce [21] and OpenAI Baselines [8], both of which are also implemented in Python using TensorFlow. However, neither library natively supports multiple action choice outputs (our components) and the ways we wanted to experiment with using them. Modifying those libraries to add that functionality appeared to be more work than creating our own implementation, which is designed with the PySC2 environment and our experiments in mind. Algorithm 6 shows our implementation, emphasizing changes unique to component-action DQN for StarCraft II. We describe the algorithm in detail in Sections 4.3.1 and 4.3.2.

4.3.1 Action Components

Q-learning algorithms work with finite discrete action spaces by outputting an action-value for each action choice at each state. Actions for PySC2 are composed of a function type and multiple parameters to most function types. The number of unique actions even in the small subset we are using is too large to output an action-value for each one (for instance, if the screen size is 84x84 pixels, there are 7056 unique "attack screen" commands alone). For comparison, the Atari Learning Environment used in the original DQN work has 17 unique actions [26]. If we expanded our approach to the full game of StarCraft II, requiring more action functions and parameters, the number of unique actions would be even larger.

To reduce the number of unique actions that need to be considered at once, we output a separate set of action-values for each action component d. We call the set of all action components D, and the set of possible actions for component d is ${}^{d}\mathcal{A}$. An action taken at time step t is $a_t = ({}^{0}a, {}^{1}a, \ldots, {}^{|D|-1}a)$. Section 4.3.3 describes how the network is branched to provide separate outputs for each action component in use. The output from the d component branch of the network is ${}^{d}Q$. Table 4.1 lists the number of choices available in PySC2 for the action function and each argument, and the number of choices we use in our experiments. As explained in Section 4.1.2, we want to limit the number of action choices to what is necessary for the scenarios we are training in. This applies to which action functions we use, as well as to which parameters we use. Some parameters can be ignored completely, such as the "queue" parameter, which allows action functions that use it to be queued. If the network is not an RNN (allowing the network to remember an action or part of the state from a previous frame), or the queued actions are not present in the state data in some

	Number of action parameters	
Action Component	Full PySC2	Our Combat Scenario Experiments
function	500+	5-6
screen	screen dimensions	$84 \cdot 84 = 7056$
screen2	screen dimensions	$84 \cdot 84 = 7056$
minimap	minimap dimensions	not using
queued	2	not using
control_group_act	5	not using
control_group_id	10	not using
select_point_act	4	not using
select_add	2	not using
select_unit_act	4	not using
select_unit_id	up to 500	not using
select_worker	4	not using
build_queue_id	up to 10	not using
unload_id	up to 500	not using

Table 4.1: PySC2 action parameter count

form, action decisions can't be based on knowledge of previously queued actions. We did not experiment with using RNN components to the network in this thesis. Additionally, queued actions would not be very useful for the small combat scenarios we use in our experiments. Therefore the only parameters we use are screen and screen2, and our network outputs 3 action components for the action function and those two parameters.

For the PySC2 environment there is a relationship between the first component, the function choice, and the other components which are parameters to the function. The choice of action for the first component (numbered 0 in Algorithm 6) determines which other components are used in the action. D_t is the set of used components for the action at time step t. Component actions in other environments may have other restrictions or relationships between components. When choosing a greedy action, the action function with the highest action-value is chosen, disregarding actions marked as unavailable in the state observation, and then each parameter to the action function is chosen according to the highest action-value for that component (step 11). An alternative method which we did not test would be to compare the average action-value of each valid function choice and its parameters if that function were chosen. This would be more costly since it requires looking up the parameters used for each valid action function plus the calculations of the averages.

Invalid action component choices are masked in several parts of the algorithm. As stated previously, action function selection in steps 7 and 11 are masked according to the valid actions for that state. Invalid choices for parameters to the action function could be masked out in steps 8 and 11 in general, but are not in this work since all values of the parameters we used (screen and screen2) are valid. Only parameters that are used by the chosen action function, ${}^{0}a$, are used in the environment update in step 12.

4.3.2 TD Target and Loss Calculations

As described in Section 3.2.3.2 the Double DQN TD target is given by:

$$y = r + \gamma \hat{Q}(s', \underset{a'}{\operatorname{argmax}} Q(s', a'; \theta); \theta^{-}).$$

With a single action component the loss to be minimized is $L(\delta) = L(y - Q(s, a; \theta))$, where $L(\delta)$ is some loss function (e.g. MSE) and δ is the TD error. When using component-actions, there are separate action-values for each component of the action, and every component is not necessarily used for every action. This raises the question of how to calculate training loss when the action components used from one action to the next differ.

Several previous works have used variations of action components with different RL algorithms. Branching Dueling Q-Network (BDQ) [45] is applied to several MuJoCo physical control tasks with multidimensional action spaces. In these tasks, all components are used in each complete action. To calculate the training loss the authors experimented with 3 formulas for the TD target: 1) using a separate TD target, ${}^{d}y$, for each action component ${}^{d}a$; 2) using the maximum ${}^{d}y$ as a single TD target; and 3) using the average of the ${}^{d}y$ as a single TD target, which was found to give the best performance:

$$y = r + \frac{\gamma}{|D|} \sum_{d \in D} {}^{d}\hat{Q}(s', \operatorname*{argmax}_{{}^{d}a'} {}^{d}Q(s', {}^{d}a'; \theta); \theta^{-}) \tag{4.1}$$

The training loss used in BDQ is the average of the MSEs of each component's action-value compared with y from Equation 4.1 [45].

Action components have also been used [16] to represent actions in the microRTS environment to train an agent using the A2C algorithm. In that work each component is not used in every action, but the policy gradient action log-probability ($\log(\pi_{\theta}(s_t, a_t)$, where π_{θ} is the policy network) used as the training loss is always the sum of the action log-probabilities for each action component. AlphaStar [50] also uses a policy gradient method and sums the action log-probabilities from each action component, but masks out the contribution from unused components.

We considered and tested three different TD target and loss calculations that are similar to the methods tested with BDQ [45], but with masking of unused components. The first was to sum the Huber losses of the components compared pairwise with

$${}^{d}y = \frac{r}{|D_{used}|} + \gamma\, {}^{d}\hat{Q}(s', \arg\max_{{}^{d}a'} {}^{d}Q(s', {}^{d}a'; \theta); \theta^{-}),$$

masking out unused components from the action taken but not from the target action. This method is similar to method 1) tested with BDQ [45]. It has the advantage of comparing losses from different action component branches of the network to their corresponding branch, but since it updates the primary network toward the action-values of unused components of the target action we didn't expect it to perform well.

The second method we tried was to use the Huber loss of the average of the action-values of the action taken by the primary network (unused components masked out) compared with the same average y from Equation 4.1. This method seems reasonable since it uses the same formula to calculate both the primary network total action-value and discounted target network next state value. However, branches with positive and negative action-values can cancel each other out in both inputs to the loss function, resulting in a small loss when individual components of the primary network action-value may have a large error relative to the target network.

The method that we found had the best performance was similar to 3) in the BDQ work. In our implementation of component-action DQN, on step 23, we use the mean target component y as in Equation 4.1 for non-terminal states, but modified so that unused components of the target action are masked using ${}^{d}m = 1$ if component d is in use for an action, ${}^{d}m = 0$ otherwise:

$$y = r + \gamma \frac{1}{\sum_{d} {}^{d}m} \sum_{d} {}^{d}m \, {}^{d}\hat{Q}\left(s', \underset{{}^{d}a'}{\operatorname{argmax}} \, {}^{d}Q(s', {}^{d}a'; \theta); \theta^{-}\right) \tag{4.2}$$

The total loss to be minimized with gradient descent on step 26 of Algorithm 6 is the sum of the losses of each used component's action-value compared to the target y:

$$L_{Total} = \sum_{d} L\left(w \, {}^{d}m(y - {}^{d}Q(s, {}^{d}a; \theta))\right),$$

which we found to perform slightly better than using the mean of those losses. w is the importance-sampling weight, and L is the Huber loss presented in Equation 3.2 in Section 3.2.3.1, which is squared error for error with absolute value less than 1, and absolute value error otherwise. In step 25 we set the new prioritized experience replay priority of a transition to be the average of the magnitude of the error across all components, with unused components masked out. Results of comparisons between the three methods for calculating loss are presented in Section 5.3.

Algorithm 6 Double DQN with prioritized experience replay and component-actions for StarCraft II

```
**Input:** discount $\gamma$, initial $\epsilon$ and $\epsilon$ annealing schedule, minibatch size $k$, batch update frequency $K$, target update frequency $C$, max steps $T$, memory size $N$, $M_{min} \leq N$, prioritized experience replay exponents $\alpha$ and $\beta$, and $\beta$ annealing schedule
Initialize replay memory $M \leftarrow \emptyset$
Initialize $Q$ with random parameters $\theta$, $\hat{Q}$ with parameters $\theta^- = \theta$
Initialize environment with state $s$
**for** $t = 1$ to $T$ **do**
    **if** rand() $< \epsilon$ **or** $t \leq M_{min}$ **then**
        Select valid action function ${}^0a$ randomly from valid action functions in $s$
        Select action parameters ${}^da$ randomly, $\forall d > 0 \in D_t$
        $a \leftarrow \{{}^0a, {}^1a, \dots\}$
    **else**
        $a \leftarrow \{{}^da = \operatorname*{argmax}_{a_t} {}^dQ(s, a_t; \theta) \mid d \in D_t\}$
    Take action $a$ in environment, observe $s', r$
    Store transition $(s, a, r, s')$ in $M$ with maximal priority $p_t = \max_i(p_i)$
    $s \leftarrow s'$
    Anneal $\epsilon$, increase $\beta$ according to schedules
    **if** $t \equiv 0 \mod C$ **then**
        Update weights $\theta^- \leftarrow \theta$
    **if** $t \equiv 0 \mod K$ **and** $t > M_{min}$ **then**
        Sample minibatch of $k$ transitions $(s_j, a_j, r_j, s'_j)$ from $M$ with $P(j) = p_j^\alpha / \sum_i p_i^\alpha$
        Compute importance sampling weights $w_j = (N \cdot P(j))^{-\beta} / \max_i(w_i)$
        ${}^0a' \leftarrow \operatorname*{argmax}_{{}^0a'} {}^0Q(s'_j, {}^0a'; \theta)$
        ${}^dm \leftarrow 1$ **if** $d = 0$ **or** $d$ is a parameter to ${}^0a'$, **else** ${}^dm \leftarrow 0$ $\forall d \in D$
        $y_j = \begin{cases} r_j & \text{if } s'_j \text{ is terminal} \\ r_j + \gamma \frac{1}{\sum_d {}^dm} \sum_d {}^dm \, {}^d\hat{Q}\left(s'_j, \operatorname*{argmax}_{{}^da'} {}^dQ(s'_j, {}^da'; \theta); \theta^-\right) & \text{otherwise} \end{cases}$
        Compute component TD errors ${}^d\delta_j = {}^dm |y_j - {}^dQ(s_j, {}^da_j; \theta)|$
        Update transition priority $p_j \leftarrow \frac{1}{\sum_d {}^dm} \sum_d {}^d\delta_j$
        Update $\theta$ with gradient descent step minimizing $\sum_d L(w_j \, {}^d\delta_j)$
```

4.3.3 Neural Network Structure

The neural network used to predict action-values consists of a shared CNN, followed by separate branches for state value and the three action components we used (function, screen, screen2). The network's overall design is shown in Figure 4.2.

The shared CNN module of the network is the largest part of the network and is where the network can learn new features from the spatial state data. In all convolutional layers we use padding and a stride of 1 to keep the output at the same width and height as the input, so as to preserve spatial data for the action parameters which target parts of the screen. Strides greater than 1 are usually used in CNNs for classification because getting a smaller output speeds up later layers. In this case we want to use the eventual output to select parts of the screen, so having the final output remain the same 84x84 avoids losing resolution in the screen and screen2 actions. For the same reason, we don't use average or max pooling in this part of the network, though those functions are common in image classification CNNs.

The design of the shared CNN is informed by two goals: 1) we would like the network to be able to learn patterns that appear in fairly large portions of the map (for instance to notice large groups of units, or the distance between groups of units); and 2) we want training times to be as fast as possible. We informally experimented with a number of designs, including very deep networks using residual blocks [11], which took too long to train. The design we use has a series of blocks of parallel convolutional layers whose outputs are concatenated. The convolutional layers in each block have different kernel sizes (decreasing as we move forward through the network), allowing features of a range of sizes to be learned in each block. Figure 4.3 shows the structure of the shared CNN module. The state input categorical feature maps are first preprocessed to be in a one-hot encoding with dimensions 84x84x6. Next the input is run through a block consisting of three parallel convolutional layers with 16 filters each and kernel sizes 7, 5, and 3. The output of those layers is concatenated along the channel dimension and given as input to a block with two parallel convolutional layers with 32 filters and kernel sizes 5 and 3. Those two outputs are concatenated and fed to a

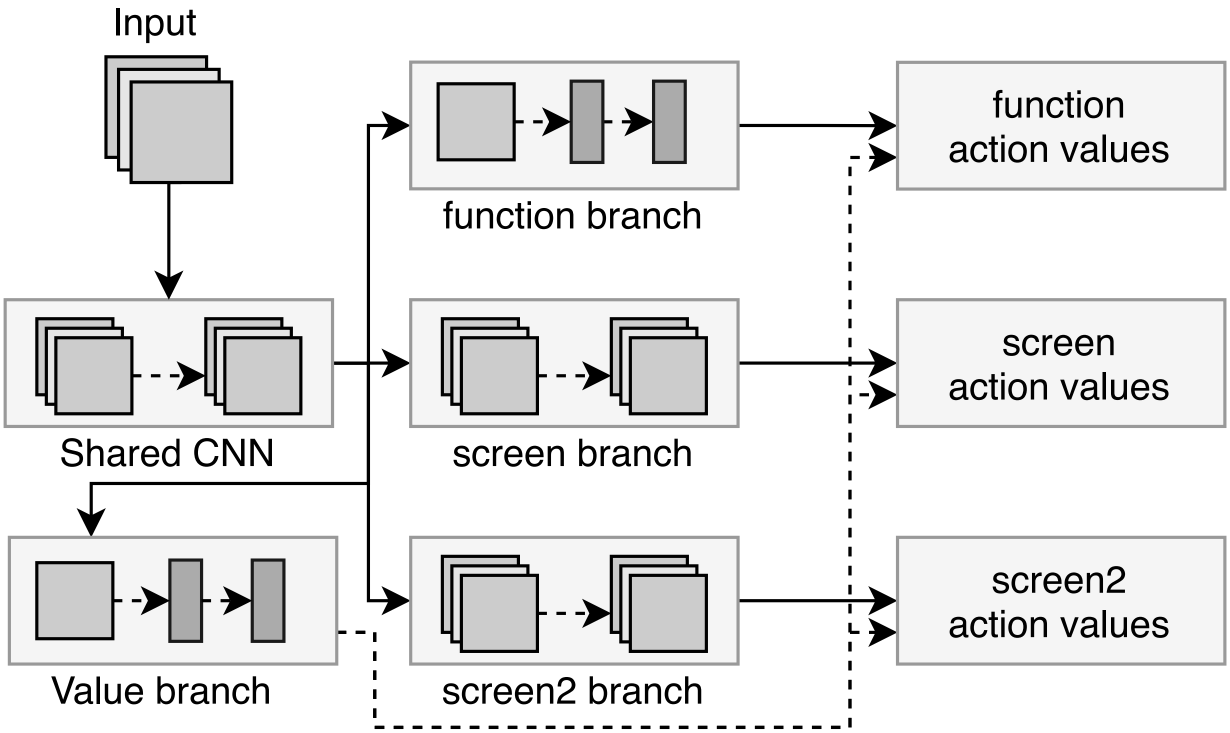


Figure 4.2: Diagram showing overall design of the network.

final convolutional layer with 32 filters and kernel size 3. The final output of the shared CNN module has dimensions 84x84x32.

Next the network splits into a value branch and one branch for each action component. Since each branch of the network propagates gradients back to the layers before the split, we scale the gradients at that point (in backprop). We scale by $1/\sqrt{2}$ for the split of the value branch with the action output part of the network, as is done in the original dueling DQN paper [52]. We also scale the action output part by $1/N$ before branching off for each action component, where N is the number of components, as in previous work featuring action components [45]. The value and function branches each have a max pooling layer of size and stride 3, followed by 2 dense layers of size 256. The function branch ends with a final dense layer outputting the action advantages of the function action component. Both the screen and screen2 branches receive as input the output of the shared CNN, which is fed into a 32 filter 3x3 convolutional layer, followed by a 1 filter 1x1 convolutional layer as described in

[51], giving output dimensions of 84x84x1 and the action advantages of each screen position.

Since the screen action component is used for multiple different action functions, it should likely output different values based on what action function will be chosen in a given state. The network structure described above forces the screen branch to essentially learn what action function will be chosen. For example, if the function is attack_screen, the screen branch should output high values for good targets to attack, whereas if the function is select_rect, it should output high values for areas of the screen near units that should be selected. Similarly, the screen2 branch should output different values based on the output of the screen branch (screen2 is only used for select_rect, when a rectangle is being specified by the outputs of the two branches). Based on this observation, we believe the network could be improved by using the output of some action component branches as input to others. AlphaStar also uses this concept in its network structure [49]. We experimented with adding the one-hot encoded output from the function branch as input to the screen branch (one 84x84 layer per function option, with the same value everywhere), followed by additional convolutional layers. Similarly, the output from the screen branch was added to the screen2 input with a single layer with a value of 1 in the position corresponding to the screen choice and 0 everywhere else. Surprisingly, we found that for the action functions and scenarios used in these experiments there was no gain in performance. We believe a larger network combined with more training time may be required to take advantage of these network connections.

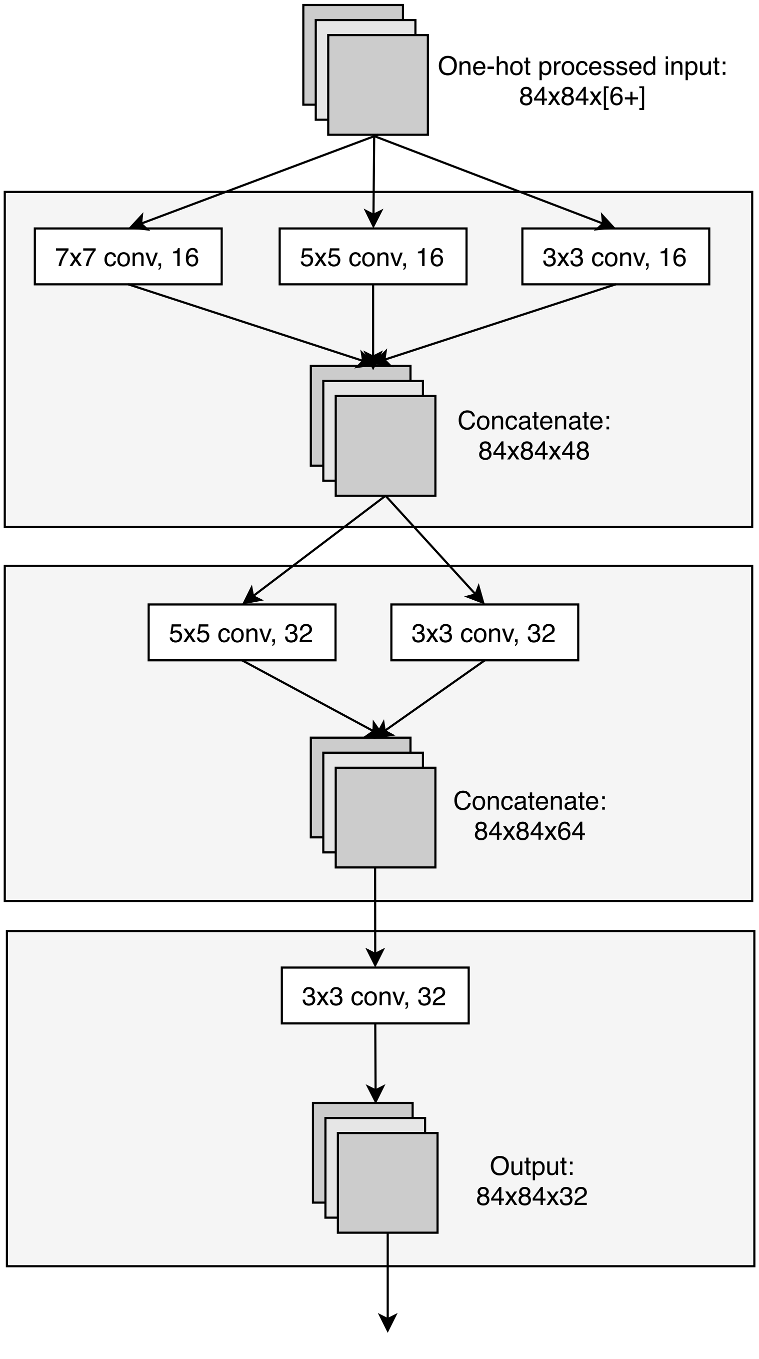


Figure 4.3: Diagram showing the shared CNN. The processed input has a different number of layers (6 or more) depending on the input features used. Convolutional filters all have stride 1, and are described as "[filter dimensions] conv, [# filters]".

4.3.4 Training Hyperparameters

Each convolutional and dense layer (except those leading to action advantage outputs) is followed by a ReLU activation and batch normalization. Training hyperparameters were selected through informal search. We use the Adam optimizer with learning rate of 0.001, a discount of 0.99, batch size of 64, and L2 regularization. Minibatch training updates are performed every 4 steps, and the target network is updated every 10,000 steps, as in the DQN work on the ALE [26]. The prioritized experience replay memory size is 60,000 transitions, and all hyperparameters for the replay memory are as described for the sum tree implementation in the paper in which it was introduced [34]. In each training run the exploration ϵ is exponentially annealed from 1 to 0.05 over the first 80% of total steps.

4.4 StarCraft II Combat Scenarios

Experiments were conducted on custom StarCraft II maps that each contain a specific combat scenario. All scenarios limit the size of the battlefield to a single-screen sized map, which removes the requirement to navigate the view to different areas of the map. The scenarios all share some common rules, but vary in the number and type of units each side has.

4.4.1 Episodes

Each scenario allows repeated episodes of play, in which an equal number of units are created for two opposing armies, placed at opposite ends of the map. An episode ends when one of the armies is defeated. In our experiments one side is always controlled through PySC2 by an RL or scripted player being trained or evaluated, while the other side is controlled by a built-in AI controlled player. The built-in AI enemy is immediately given an order to attack the opposite side of the map, causing them to rush at the PySC2 controlled player's units, attacking the first enemies they reach. Once given this command, the in-game AI takes over control of the enemy units, which executes a policy which prioritizes attacking the closest units of the RL player. This use of the built-in AI to control the opponent for combat experiments has been shown to be effective for testing and training combat algorithm development in previous works [6].

In all scenarios units appear in two randomized clusters, randomly assigned to the left or right side of the map. The clusters are symmetric about the diagonal line going form the bottom-left to top-right corners of the map. These scenarios were constructed in a symmetric fashion in order to give both sides an equal chance at winning each battle. If one side is victorious from an even starting position, it must mean that their method is more effective at controlling units for combat. One effective policy for such scenarios could be to first group the players' units into a formation that will allow all of the units to attack the same enemy, and then using focus-targeting to most efficiently destroy enemy units.

There are no buildings or *worker* units present in any scenarios. Workers could be used for harvesting resources and constructing buildings. There is also no terrain variability in any of the maps. The scenarios are designed for learning and evaluating policies for combat between small groups of combat units in a small map area.

Episodes end when all units of one side are destroyed (health reduced to 0), or when a timer runs out. The timer is set to a value between 45 and 90 real-time game seconds (meaning it would run for 45 to 90 seconds if the game were running at standard human-play speed), depending on the units used in the scenario. Some units used, such as the Protoss Stalker, have a higher amount of health and shields but similar amount of damage dealing capability, so they take longer to be destroyed. If the timer is set appropriately high for the number and type of units in the battle, the timer will only run out if an RL player learns to send units into a corner and keep them there, where the enemy units don't see them. When the episode is over, the map then resets with a new randomized unit configuration and a new episode begins.

4.4.2 Rewards

RL algorithms depend on a reward signal to indicate if an action taken in a state was a good or bad one. Maximizing future reward is the goal of the algorithm when adjusting the policy. In adversarial games such as StarCraft II the true goal of a player is to win, and the *closeness* of the win doesn't matter. Therefore, the reward signal which would represent the least amount of hand-crafting with human knowledge would be to award +1 if the episode ends in a win, -1 for a loss, and 0 for a draw. That would be a sparse reward since it is given only at the end of the episode. The model learns an approximation of the value of being in a given state, which is the expected future reward, so it's possible to learn an effective policy from only a sparse end-of-episode reward, but the learning time will be longer than if we use a more frequent reward signal.

For our scenarios we chose to use a reward function that gives the RL player a more direct and immediate indication of it's progress during an episode. Some elements that could be used in a reward function for these combat scenarios are unit health, number of units remaining, and unit positions. [5] describes two evaluation formulas for combat games previously shown to work well as tree search evaluation functions. The first is LTD (life-time damage), which values the units in a sate s proportionally to their remaining health (hp) multiplied by their damage-per-frame:

$$\mathrm{LTD2}(s) = \sum_{u \in U_1} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \mathrm{hp}(u) \cdot \mathrm{dpf}(u),$$

where units $u \in U_p$ belong to player p, $\mathrm{hp}(u)$ is the remaining health of unit u, and $\mathrm{dpf}(u)$ is the damage per frame of unit u, calculated by dividing the damage inflicted by one attack from unit u by the weapon cooldown time (the time in frames the unit must wait before attacking again) of unit u.

The second formula is LTD2, which uses the square root of the units' remaining health:

$$\mathrm{LTD2}(s) = \sum_{u \in U_1} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u)$$

We chose to use LTD2 as our reward formula because it offers the most useful information for RL. LTD2 favours higher health for the most damaged friendly unit over less damaged ones, or lower health for the most damaged enemies over less damaged ones, i.e. it is more valuable to damage an enemy unit that is closer to being destroyed. LTD2 conveys a basic intuition about RTS combat, which is that it is better to focus on enemies one or a few at a time to remove their ability to cause damage, rather than attack all enemies equally at the same time. If we used the LTD formula, the RL player would receive the same reward if it equally damaged a full health enemy or a nearly destroyed enemy. The $\mathrm{dpf}(u)$ parts of the formula give the RL player explicit feedback that more powerful units are more valuable. In some of our scenarios, described in Section 4.4.3, units have shield values in addition to health, so the LTD2 formula is modified so that $\mathrm{hp}(u)$ gives the sum of the shield and health values of unit u.

Our custom StarCraft II maps output reward information as a score which is read by PySC2 and observed by the RL player in step 12 of Algorithm 6. On every frame that a unit has been damaged since the last frame, the new LTD2 value is calculated and the difference from the previous frame is output as a reward. Rewards can be positive or negative, and if units of both sides are damaged between two frames the reward output may be 0. Rewards are normalized to equal 1 for destroying a full health unit of the highest LTD2 value in the scenario (only relevant if multiple unit types are present in a scenario). Since there are positive rewards for damaging enemy units and negative rewards for taking damage, episode rewards tend to be similar in scenarios with different numbers of units.

Identifier	Unit Type(s)	Unit Counts per Player	Training Feature Layers (Count)
nm vs. nm	Marines	$n \in \{4, 8, 16, 32\}$	player_relative (2) selected (1) unit_hit_points (3)
4-32m vs. 4-32m	Marines	$4 \leq n \leq 32$	player_relative (2) selected (1) unit_hit_points (3)
ns vs. ns	Stalkers	$n \in \{4, 8, 16, 32\}$	player_relative (2) selected (1) unit_hit_points (4) unit_shields (4)
nh vs. nh	Hydralisks	$n \in \{4, 8, 16, 32\}$	player_relative (2) selected (1) unit_hit_points (4)
nz vs. nz	Zerglings	$n \in \{4, 8, 16, 32\}$	player_relative (2) selected (1) unit_hit_points (3)

Table 4.2: All scenarios used in experiments.

4.4.3 Scenarios

We created combat scenarios, custom StarCraft II maps with scripted triggers, using a variety of unit types and counts. Table 4.2 lists all of the scenarios we used in our experiments, and Table 4.3 lists the relevant properties of the units used in the scenarios we created. Unit counts range from 4 to 32. In the 4-32m vs. 4-32m scenario the unit count per side is a new random integer between 4 and 32, inclusive, for each episode. We used units from the three races in StarCraft II, Terran, Protoss, and Zerg. The only difference between those races which affects the scenarios we use is that all Protoss units have shields. Shields, like health, are depleted by damage but must be lowered to 0 before a unit's health can be lowered. Shields also replenish slowly over time. The Protoss Stalker is also the only unit with armour, which reduces incoming damage.

We use three ranged unit types and one melee unit type in our experiments. Terran Marines are ranged units with relatively small health and damage. The Protoss Stalker is

Type	Health	Shields	Armour	DPS	Range	Size	Speed
Terran Marine	45	0	0	9.8	5	0.75	3.15
Protoss Stalker	80	80	1	9.7	6	1.25	4.13
Zerg Hydralisk	90	0	0	22.2	5	1.25	3.15
Zerg Zergling	35	0	0	10	-	0.75	4.13

Table 4.3: Properties of all units used in combat scenarios, as compiled on the Liquipedia StarCraft II wiki [23]. DPS is damage per second, the amount of damage the unit inflicts per second of gameplay. Speed is measured in game distance units per second. Range and size are measured in game distance units. A unit's size determines its collisions with other units. For comparison, all scenarios have a map size of 22x16 game distance units.

a larger ranged unit with slightly longer range, more health, shields, and armour. Because its damage output is almost the same as the Marine's, but it takes much longer to destroy Stalkers and their shields can replenish, it's a more viable strategy in a Stalker scenario to move units from the "front lines" to the back so that the enemy will start attacking other units with full health and shields. The 16s vs. 16s scenario is shown in Figure 4.4. Zerg Hydralisks are another larger ranged unit that have double the health of Marines and 2.27 times their damage output, so in comparison to Marine scenarios Hydralisk scenarios should end faster. Zerg Zerglings are a small, fast, melee unit without a lot of health. Optimal positioning should be very different in Zergling scenarios compared to scenarios featuring the other ranged units.

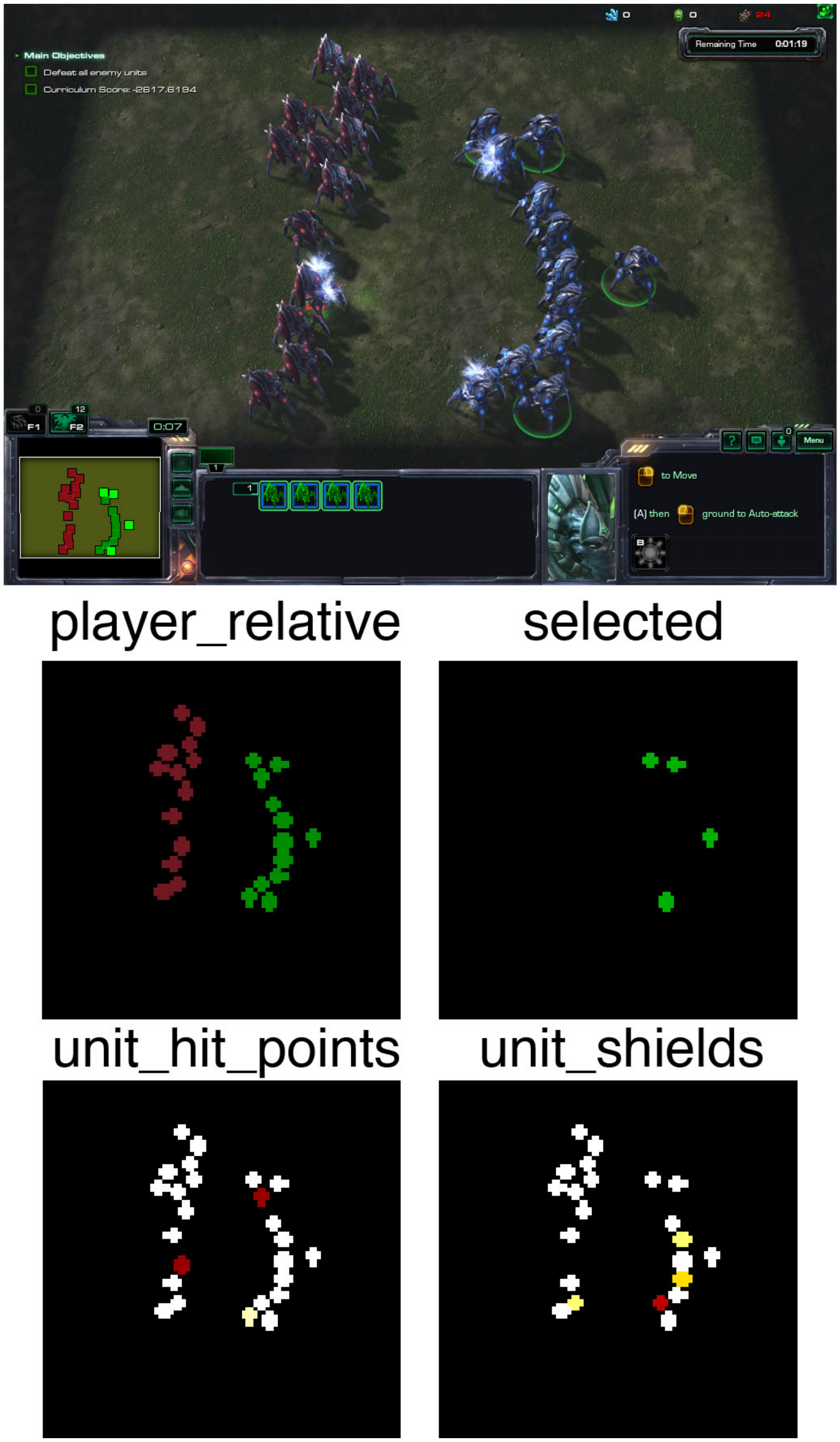


Figure 4.4: 16s vs. 16s scenario mid-episode with screenshot of game interface on top and visualization of feature layers, labelled, on bottom. In the unit_hit_points and unit_shields features values range from full to low with the colours white, yellow, orange, and red.

Chapter 5

Experimental Results and Discussion

In this chapter we present the results of a number of experiments which test the performance of component-action DQN with our network structure and action representation in microRTS and StarCraft II combat scenarios, and the ability of trained models to maintain performance when transferred to other scenarios with and without additional training. In Section 5.1 we discuss the results of our experiments using the microRTS environment. In Section 5.2 we detail how we conducted our PySC2 experiments and the evaluation metrics used. In Section 5.3 we compare the performance of methods for calculating the TD target and loss. In Section 5.4 we show the performance of four simple scripted players we created in various StarCraft II combat scenarios playing against the game's built-in AI. These scripted players follow programmed rules such as "attack the nearest enemy".

Our main results in training RL models in the PySC2 environment are presented in Section 5.5. In Section 5.5.1 we discuss the affects of training time on performance. In Section 5.5.3 we discuss the types of policies the trained models learned. In Sections 5.5.4 and 5.5.5 we present the results from transferring learning across different battle sizes and unit types, respectively. In Section 5.5.6 we show the results of training on a scenario with a random battle size. We also trained models by changing the scenario during training time, which we discuss in Section 5.5.7.

5.1 microRTS Experiments

As explained in Section 4.2, although not the main focus of this thesis, we spent some time applying our DQN implementation for component-action RTS games to the AI testbed microRTS. We conducted experiments training an agent for up to 10m steps, competing against a range of bots included with microRTS which employ either simple scripted behaviours or search algorithms such as MCTS. In each experiment we cycled through different 8x8 sized maps. We also experimented with self-play during training, since the scripted and search-based bots are very effective on small maps and we hoped that playing against a bot at the same level as the RL agent would help with training. Most maps included bases and workers, so the task was in a way more complex than the combat scenarios we are using in PySC2, despite being played on smaller maps.

We were never able to see what we considered good results with this training. The best policies that we saw from trained models were to build workers with the limited starting resources and then do something like a "worker rush", a common RTS strategy where a player attempts to win as early as possible using only the units available at the start of a match. Even this strategy was inconsistently applied, with many seemingly random moves despite evaluating with exploration ϵ set to a low value or zero.

We believe there were two main reasons this approach to playing microRTS was not successful. The first is that the system we devised for formatting actions was too complex. In order to make good decisions the branches of the network for each component would essentially have to learn what's being learned in the select component branch, since the unit-action to take depends heavily on which unit is acting. One way to account for this is to have the output of the select branch be fed back in to the other branches, which we did not explore while experimenting with microRTS. The second issue with our approach was that the scenarios we started with were too difficult, as they comprised the full game of microRTS, despite being small in size. We believe success would have been more likely if we had used scenarios with more limited actions available, such as in [16]. After finding poor results in this environment we focused fully on PySC2 research, but we believe that with some modifications a version of our approach could be used in microRTS to more quickly test RL performance in RTS games.

5.2 PySC2 Training and Evaluation

All trained models are trained on machines with an Intel i7-7700K CPU running at 4.2 GHz and an NVIDIA GeForce GTX 1080 Ti video card. It takes 5 hours to train for 300k steps using the GPU and running the game as fast as possible. To train a model we use PySC2 to run our custom combat scenarios with our RL player. The RL player receives input from the game environment and takes actions as described in Algorithm 6. In all cases training was ended after a set number of steps is reached. Because each episode can play out differently the number of episodes per training run using the same step limit can vary.

Throughout this chapter we present results of evaluation of trained policies and scripted bots, described in Section 5.4, in different combat scenarios. Trained models are evaluated with a deterministic policy (i.e. $\epsilon = 0$), using the trained model for inference only. Scripted bots follow rules-based policies, which may include randomness. All training and evaluation is done with the opponent being the game's built-in AI, i.e. the same opponent a human player playing the single-player game would face. In evaluation, the result can be a win (1 point), draw (0.5) or loss (0). Wins occur when all enemy units are destroyed. Draws happen if both sides simultaneously lose their last unit. If the map timer runs out, the side with the most combined health/shields wins. If that metric results in a tie then the episode is counted as a draw. Evaluation results are presented as a score, equal to the number of wins plus half of the number of draws, divided by the number of evaluation episodes.

5.3 Comparison of Loss Formulas in Component-Action DQN

As described in Section 4.3.2, we compared three methods for calculating the TD target and loss in component-action DQN training updates. The three methods we tested are: 1) comparing each component of the primary and target networks pairwise; 2) comparing the average target y to each used component of the primary network; and 3) comparing the average target y to the average action value of the components used in the primary network. In all methods unused components are masked out from the calculations, except in the case of the target network in pairwise comparison. For that method all used components in the primary network need to be updated, so the corresponding target network component output is used in the calculation even if it is not used for the target greedy action. In all cases the loss is calculated across a batch of size 64 in our experiments.

We trained 10 models for each of the the three methods for 100k steps on the 8m vs. 8m scenario. The episode training rewards can be seen in Figure 5.1. Table 5.1 gives the scores after 1k episodes of evaluation for each of the three methods. The best performing models for all three methods perform similarly, with the second method, which compares the average target y to the average of the components in the primary network, scoring the highest. The third method, which compares the average target y to each component in the primary network, has the highest average score. Similarly, the episode training rewards in Figure 5.1 show that the models trained with the third method have the highest ending rewards, on average. We chose to use this method in the rest of our experiments because of its high average performance.

Loss Formula	Best Score	Average Score
Pairwise Comparison	0.630	0.318
Avg. y compared to avg. prediction	**0.715**	0.205
Avg. y compared to components	0.686	**0.424**

Table 5.1: Best and average scores on 1k episodes of evaluation of 10 models trained with each of 3 different loss formulas. Highest best and average scores are in bold.

5.4 PySC2 Scripted Player Benchmarks

We created several simple scripted players which represent strategies that we may expect the trained models to match or exceed, depending on the scenario. The scripted players are:

No Action (NA) - This player takes the no_op action only, which is equivalent to a human player providing no input. Its units will attack and follow enemy units if they move into range, which is controlled by the in-game AI.

Random (R) - This player takes random actions. Both the function choice and any arguments are randomized.

Random Attack (RA) - This player selects all friendly units, and subsequently attacks random screen positions.

Attack Weakest Nearest (AWN) - This player selects all friendly units on the first frame and then attacks the enemy with the lowest health, choosing the enemy nearest to the average of the friendly units' positions as a tie breaker.

We evaluated each scripted player for 1k episodes on scenarios featuring Marines, Stalkers, Hydralisks, and Zerglings, with 4, 8, 16, and 32 units per side, which are described in Section 4.4.3. The results of the scripted players evaluation can be seen in table 5.2. The performance of the scripted players shows a large range of results, differing with the number and type of units, as well as with policy. We expected the AWN player to achieve the highest scores in most scenarios, but found that other policies perform better depending on the scenario properties, particularly with larger unit counts.

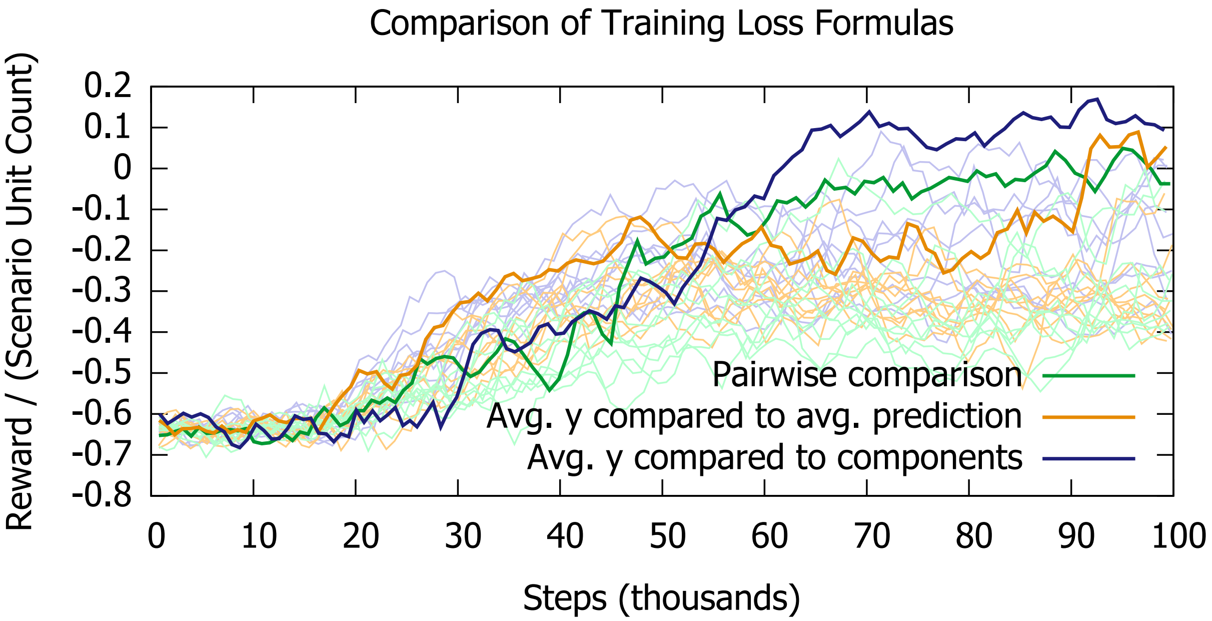


Figure 5.1: Training episode reward on 8m vs. 8m (normalized to the number of Marines per side in the scenario) per episode (smoothed over 100 episodes) for 10 runs each with three loss calculation methods. Lines in bold are the top ending reward for each method. Faded lines in corresponding lighter shade are other models trained with the same method.

In general the scripted players perform best in the 4 vs. 4 scenarios and perform worse as the number of units increases. The only exception is for the No Action policy on the Stalker and Hydralisk scenarios, which performs better (and in fact better than every other scripted policy) in the scenarios with 16 and 32 units per side. Taking no actions means that units will attack when enemies get within range, so it's not surprising that the strategy sometimes wins, depending on the randomized starting positions of units. For example, if the scripted player's units are more spread out, it would likely perform worse because the full group of enemy units will attack the scripted player's units in several smaller groups at a time, and thus have an advantage over the scripted player. We believe the NA player sees better performance in the Stalker and Hydralisk scenarios at higher unit counts because those unit types have higher health and survive for longer in battles. This gives nearby units longer to become engaged in fighting before the first units are killed by the enemy built-in

Evaluation	Scripted Players			
Scenario	NA	R	RA	AWN
4m vs. 4m	0.173	0.013	0.570	**0.907**
8m vs. 8m	0.146	0.019	0.561	**0.721**
16m vs. 16m	0.119	0.004	**0.391**	0.038
32m vs. 32m	0.044	0.002	**0.116**	0.000
4s vs. 4s	0.142	0.000	0.352	**0.910**
8s vs. 8s	0.179	0.000	0.240	**0.678**
16s vs. 16s	**0.211**	0.001	0.030	0.013
32s vs. 32s	**0.130**	0.000	0.002	0.000
4h vs. 4h	0.160	0.007	0.464	**0.705**
8h vs. 8h	0.223	0.005	**0.379**	0.100
16h vs. 16h	**0.218**	0.003	0.100	0.000
32h vs. 32h	**0.101**	0.000	0.039	0.000
4z vs. 4z	0.089	0.033	**0.562**	0.137
8z vs. 8z	0.032	0.019	**0.527**	0.001
16z vs. 16z	0.011	0.005	**0.426**	0.000
32z vs. 32z	0.001	0.000	**0.271**	0.000

Table 5.2: Scores ((wins+draws/2)/# episodes) of scripted players evaluated for 1k episodes in all scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the AWN column, 8m vs. 8m row gives the score of the AWN scripted player when evaluated on the 8m vs. 8m scenario. The best performing policy for each scenario is bolded.

AI. This is in contrast to the Zergling scenarios, for example, since Zerglings are very low health and die quickly. At higher unit counts the negative effect of starting more spread out in some episode initializations is also reduced.

The random player gets very low scores ranging from 0 to 0.033 across all scenarios. This is expected since completely random actions will result in many move actions with random selections of units. Move actions are unhelpful while in the middle of combat because units stop firing while executing a move. The random attack player achieves the highest score in the larger Marine scenarios, the 8h vs. 8h scenario, and all of the Zergling scenarios. Attacking randomly is not a bad strategy in general because attacking a point on the ground (with no unit present) causes the units to move towards that point while attacking any enemy in range. While doing that, the units of the RA player move closer together. In larger scenarios the RA player could sometimes target it's own units, causing self-inflicted damage, and it may target enemy units far away from its own units, causing them to move towards that unit without attacking any other enemies.

AWN achieves scores of 0.907, 0.910, and 0.705 in the 4 vs. 4 Marine, Stalker, and Hydralisk scenarios, respectively, but scores 0 in all 32 vs. 32 scenarios. This indicates that focus firing alone is powerful in scenarios with smaller numbers of untis, but some other strategy is required to perform well with more units. AWN does not move its units into any kind of formation, which may be a bigger factor in scenarios with more units. We observed that units often get stuck while trying to move to attack the same target, i.e. all the units of the AWN player cannot reach the targeted enemy. AWN does not perform well in the Zergling scenarios, which is likely due to that being the only scenario with melee units. Focus firing on individual units with a large group of melee units results in some of the attackers running around its allies trying and failing to get into position to attack.

The performance of these scripted players shows that no one of these strategies is best for all scenarios, and that each one has weaknesses. Focus firing, which AWN does, is very strong for small groups of units, which may indicate it is a good strategy to switch to partway though a scenario with more starting units. The difference in performance for the NA, RA, and AWN bots in the Zergling scenarios when compared to the other 3 scenarios indicates that ranged and melee units should require different strategies.

5.5 Component-Action DQN Performance in StarCraft II Combat Scenarios

We trained models on all the scenarios listed in Section 4.4.3, and evaluated those models on those scenarios as well as others to test transfer learning between scenarios. Specifically to test both the performance of transfer learning and the effects of training time, we trained models for 300k and 600k steps on Marine scenarios with 4, 8, 16, and 32 units per side, and

	Learned Policy - 300k Steps				Learned Policy - 600k Steps			
Evaluation	Trained on Marines				Trained on Marines			
Scenario	4m	8m	16m	32m	4m	8m	16m	32m
4m vs. 4m	**0.691**	0.661	0.619	0.625	**0.663**	0.606	0.444	0.414
8m vs. 8m	**0.694**	0.691	0.692	0.541	**0.693**	0.585	0.356	0.440
16m vs. 16m	0.591	0.546	**0.695**	0.450	**0.643**	0.213	0.300	0.503
32m vs. 32m	0.329	0.280	**0.441**	0.332	0.340	0.247	0.130	**0.471**
4s vs. 4s	**0.497**	0.491	0.008	0.479	**0.525**	0.291	0.236	0.408
8s vs. 8s	**0.420**	0.406	0.015	0.386	**0.512**	0.192	0.204	0.303
16s vs. 16s	0.114	0.183	0.034	**0.356**	**0.288**	0.152	0.154	0.196
32s vs. 32s	0.075	0.239	0.223	**0.320**	**0.386**	0.193	0.355	0.170
4h vs. 4h	0.543	**0.575**	0.059	0.450	**0.600**	0.348	0.282	0.334
8h vs. 8h	0.484	**0.529**	0.084	0.451	**0.615**	0.319	0.247	0.331
16h vs. 16h	0.271	0.346	0.078	**0.349**	**0.439**	0.144	0.166	0.355
32h vs. 32h	0.323	**0.505**	0.083	0.242	**0.549**	0.281	0.351	0.235
4z vs. 4z	0.535	**0.586**	0.431	0.417	**0.507**	0.459	0.455	0.409
8z vs. 8z	0.533	**0.543**	0.365	0.406	**0.506**	0.443	0.360	0.345
16z vs. 16z	0.503	**0.528**	0.322	0.309	0.381	**0.410**	0.356	0.288
32z vs. 32z	0.447	**0.479**	0.158	0.214	0.290	0.257	**0.408**	0.245

Table 5.3: Experiment scores ((wins + draws/2)/# episodes) of learned policies evaluated for 1k episodes in multiple scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the 300k 4m column, 8m vs. 8m row gives the score of the model trained for 300k steps on the 4m vs. 4m scenario when evaluated on the 8m vs. 8m scenario. The best performing policy in each category (trained for 300k steps, and 600k steps) per scenario is in bold.

then evaluated those learned models on Marine, Stalker, Hydralisk, and Zergling scenarios. For each scenario and step count combination we trained three models and used the best performing of those for the transfer learning experiments. Results from those evaluations are presented in Table 5.3. The first column shows the scenario for which the experiment is being conducted along a row, with the other column values showing which policy controls the player units for the experiment. We refer to that table in the following sections while discussing training time, transfer learning, and the effects of scenario size (unit count).

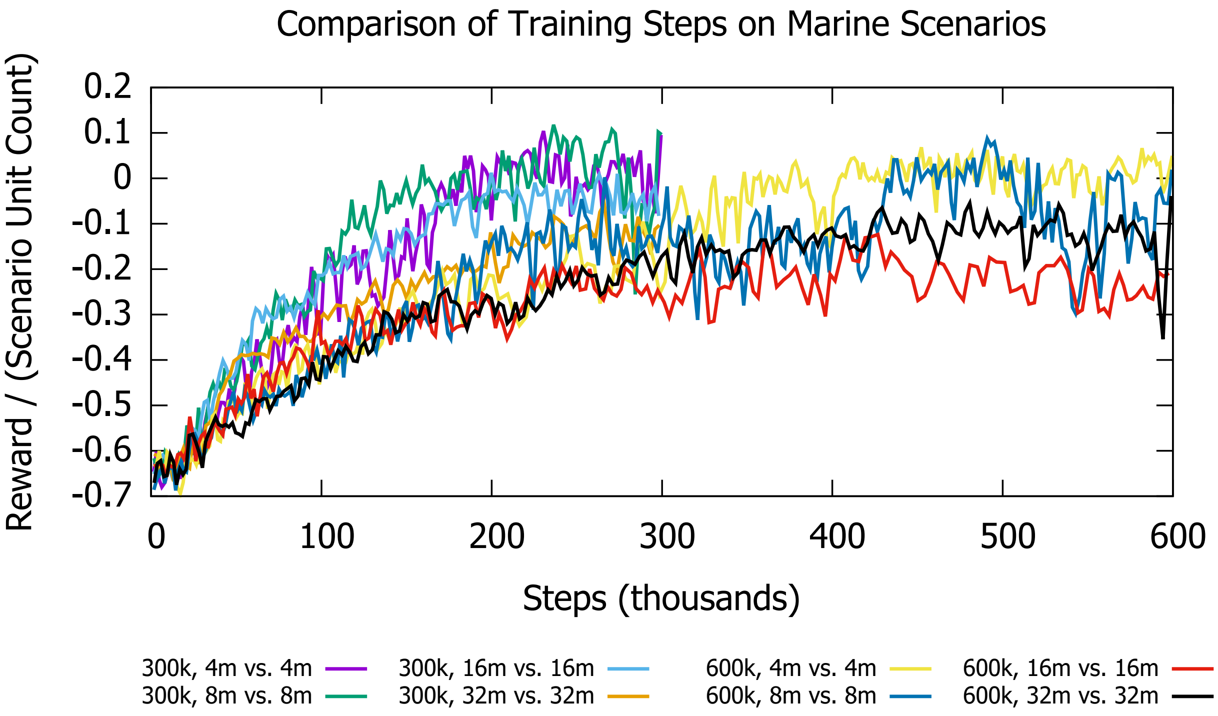


Figure 5.2: Model training reward (normalized to the number of Marines in the scenario) per episode (smoothed over 100 episodes) for 300k and 600k training steps on Marine scenarios with 4 to 32 units per side.

5.5.1 Training Time

In informal testing we found that 300k steps was enough to see good performance in our StarCraft II combat scenarios, and we chose double the number of steps as a longer training time for comparison. Normalized training rewards per episode for 300k and 600k steps in Marine scenarios with 4, 8, 16, and 32 units per side are shown in Figure 5.2. The end of training rewards are higher for the 300k step models when compared to their corresponding 600k step models. Additionally, variability in performance is higher near the end of training for the 600k step models.

Table 5.3 shows comparisons of the final models when evaluated for 1k episodes. A surprising result was that most of the policies trained for 300k time steps ended up performing as good, or even better than those trained for 600k time steps. When looking only at the performance of models in the same scenarios they were trained on, the 300k step models outperform the 600k step models in all but the 32m vs. 32m scenario. The model trained on the 4m vs. 4m scenario for 600k steps does perform best in many of the Stalker and Hydralisk scenarios, but given that models trained on other scenarios for 600k steps don't perform comparably, we think it it more likely the result of chance than a result of being trained for more steps.

There are several possible reasons that we don't see an improvement with 600k steps of training over 300k steps. One is that the scenarios may not be complicated enough for more training to result in better policies. We think this is unlikely because we know from human play that better performance is possible in these scenarios. It is also possible that the variability in model performance is too large to see a trend for the number of models we trained. Another reason that we may not see better performance with longer training times in this experiment is that we do not grow the replay memory size with the length of training. As detailed in 4.3.4, while we do scale some other hyperparameters with the training time, such as the exploration (ϵ annealing) schedule, the size of the replay memory is limited by available system RAM. The replay memory holds transitions from the last N steps, with $N = 60000$ in these experiments. Near the end of training, the exploration rate is low and there is a lower chance to leave a local maximum in the policy space. If N were higher, the model could continue to learn from transitions that were seen further back in the training, which might lead to promising new policies.

5.5.2 Comparison to Scripted Players

The trained models presented in Table 5.3 start their training with a similar policy as the random scripted player, and almost all of them outperform both the random (R) and no action (NA) scripted players by the end of training. This is a minimal indication that the models are learning and moving towards better performance as they are trained. In fact, most trained models are also able to outperform all of the scripted players at larger scenario sizes featuring 16 or 32 units per side. The random attack (RA) scripted player has comparable performance to the best trained models in the smaller Zergling scenarios, and the attack weakest nearest (AWN) player outperforms the best trained models by a larger margin in the smaller Marine and Stalker scenarios and the 4h vs. 4h scenario.

It is expected that on the these small scenarios with ranged units the scripted behaviour of targeting a single enemy unit at a time would be the best policy. That the trained models cannot match this performance indicates that such precise targeting is difficult for the RL player to learn with this combination of algorithm, state representation, and action representation. Conversely, in larger scenarios a less precise strategy may be more effective, as indicated by the scores of the RA scripted player and the trained models.

5.5.3 Learned Policies

By visually observing the real-time performance of the trained models, we can see that most models trained for 300k-600k steps learn to select all friendly units using the *select_army* command. Following that, most of those models will perform attack actions most or all of the time. Some models learn to exclusively target the ground near the cluster of friendly units, causing those units to move close together and attack enemies as soon as they are in range. We have also observed models that learn to target near the cluster of enemy units, resulting in similar behaviour but with its own units not moving as close together. Some models have also learned to focus fire on damaged enemy units by targeting them. Notably they tend to learn to target enemies that are already damaged instead of always targeting one enemy, as the AWN scripted player does. This strategy may be more effective in scenarios with more units as targeting the ground causes less wasted moving time. We have also seen policies that change their selection during the episode using select_rect, but those policies tend not to perform well.

Figure 5.3 shows the frequency of action function choice per episode during training of the best performing 8m vs. 8m model over 300k steps. It shows that use of attack_screen increases throughout training with several temporary decreases, while use of move_screen

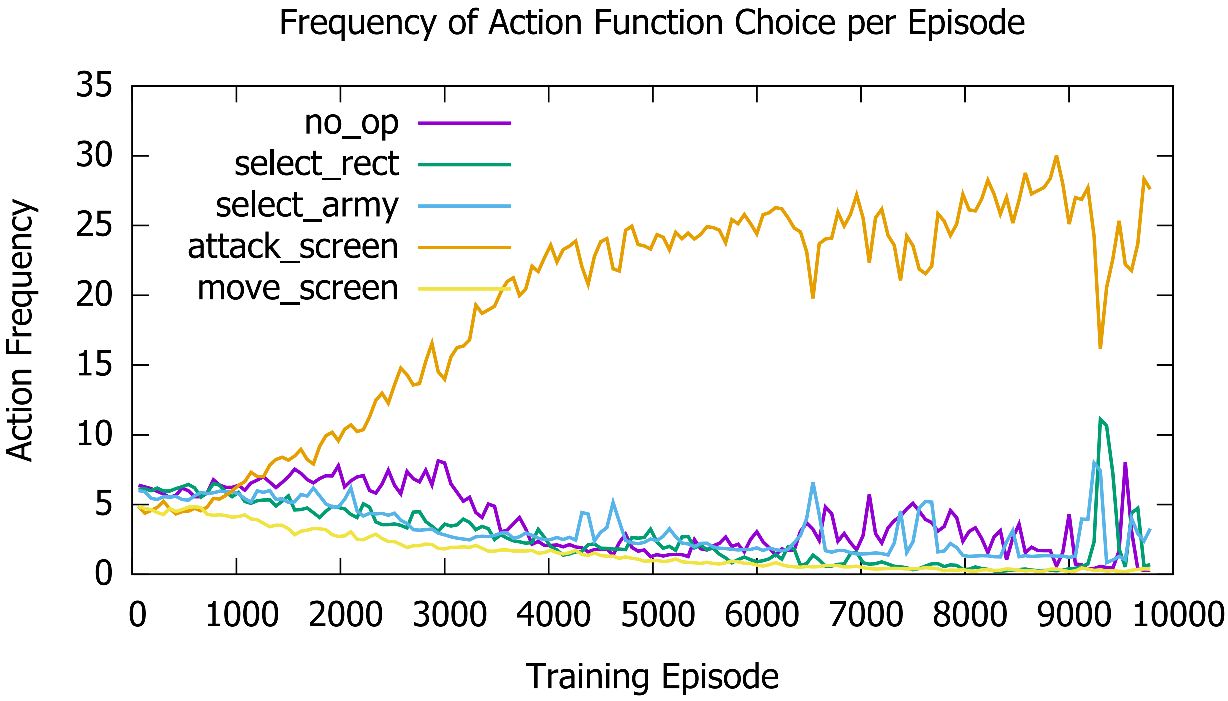


Figure 5.3: Action function frequency per episode over 300k training steps on the 8m vs. 8m scenario, smoothed over a 100 episode window.

steadily decreases. Usage of no_op, select_rect, and select_army decrease over time with occasional increases as the model explores other policies. Since the chance of random actions never goes below 5% ($\epsilon = 0.05$) during training, no action frequencies ever go to zero. It is also worth noting that if the policy is to select all the units once and then attack, then select_army only needs to be chosen once at the beginning of the episode. If a model learns to change its selection several times throughout an episode, the frequency of select_army and select_rect will still be low compared to that of attack_screen. Therefore most trained models have a similar looking action function frequency graph.

One poor and initially confusing policy we have seen is to attack at coordinates $(0, 0)$ exclusively. By investigating the network output in this case we found that this occurs when the model learns to output identical action-values for every screen coordinate in the *screen1* component, which controls attack targeting. According to that output, there is no advantage to choosing any one screen coordinate over any other. $(0, 0)$ is chosen because the max()

function used chooses the first option when there is a tie.

5.5.4 Battle Size Transfer

Results for battle size transfer for models trained on Marine scenarios can be seen in Table 5.3 by comparing the values in a single column within a group of scenarios featuring the same unit (e.g. all Marine scenarios). We believe that these results yield two major observations. The first is that, like scripted players, smaller sized battles tended to yield higher scores for learned policies across all the unit types tested (though not in every case). We believe this is because smaller battles are less complex, with far fewer possible states, and therefore are easier for learning effective policies. Also, smaller battles end faster, so more battles are able to be carried out in the same number of training steps. In both the 300k step models and the 600k step models, the ones trained on 16 or 32 unit scenarios are rarely the best performing model when considering all scenarios. For example, the model trained on 4m vs. 4m for 600k steps has the best performance compared to other 600k models for all but three of the scenarios.

The second observation was that the experiments successfully demonstrated the ability to transfer a policy trained on battles of one given size to another, while maintaining similar results. For example, a policy trained on 4 vs. 4 Marines for 600k time steps was able to obtain a score of 0.663 when applied to the 4 vs. 4 Marine scenario, and 0.643 when applied to the 16 vs. 16 Marines scenario, which was even better than the policy which was trained on 16 Marines. We did however notice that results get worse as the difference in unit count between training and testing scenarios gets larger, which was expected. One surprising result however was that for several scenarios, the best policy was not one that was trained on that same scenario. For example, the policy trained on 16 vs. 16 Marines for 600k time steps was the 2nd worst policy when applied in the 16 vs. 16 Marine scenario.

5.5.5 Unit Type Transfer

The results in the bottom 12 rows of Table 5.3 are for experiments carried out with models trained on scenarios with Marine vs. Marine battles, but evaluated on scenarios with Stalkers, Hydralisks, and Zerglings. In this section we compare the results from those models with models trained on Stalker, Hydralisk, and Zergling scenarios for 300k steps. We find that, in some cases, models trained on scenarios with a different unit type than the one used in the scenario outperform models trained on that same scenario.

Results of models trained on Stalker scenarios and Marine scenarios and evaluated on Stalker scenarios are presented in Table 5.4. They strongly suggest that it is difficult to learn good policies while training on Stalker scenarios using the same network structure and training hyperparameters as for the Marine scenarios. The best scores per evaluation scenario for the models trained on Stalker scenarios range from 0.144 (32s) to 0.227 (16s), as compared to 0.320 (32s) to 0.497 (4s) for the models trained on Marine scenarios. The policies the models are learning tend to avoid using attack actions, instead performing no_op and select_rect actions almost all the time. They are essentially learning to do nothing (since selecting and then not attacking or moving is equivalent to taking no action), which is why these scores are similar to those of the NA scripted player in Table 5.2.

We were not able to identify which aspect of the Stalker scenarios is leading to the poor training results, but there are a number of possible explanations we think are likely. Stalkers have shields in addition to health, and the shield value is included in the state representation when training on Stalker scenarios with 3 extra feature layers. When training on the Stalker scenarios the reward signal also includes the shield value as well as health, which we expected would lead to better performance relative to the models trained only on Marine scenarios without access to the shield data in the reward signal and state input. Since the state input is larger, one reason for the poor performance may be that the the model requires a larger shared CNN part of the network to learn the necessary features based on the input. We experimented informally with adding more filters to the convolutional layers in that part of

	Learned Policy - 300k Steps				Learned Policy - 300k Steps			
Evaluation	Trained on Marines				Trained on Stalkers			
Scenario	4m	8m	16m	32m	4s	8s	16s	32s
4s vs. 4s	**0.497**	0.491	0.008	0.479	**0.151**	0.141	0.138	0.138
8s vs. 8s	**0.420**	0.406	0.015	0.386	0.178	**0.212**	0.193	0.180
16s vs. 16s	0.114	0.183	0.034	**0.356**	0.221	0.218	**0.227**	0.219
32s vs. 32s	0.075	0.239	0.223	**0.320**	0.129	0.120	0.118	**0.144**

Table 5.4: Experiment scores ((wins + draws/2)/# episodes) of policies trained on Marine and Stalker scenarios, evaluated for 1k episodes in Stalker scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the 300k 4m column, 8s vs. 8s row gives the score of the model trained for 300k steps on the 4m vs. 4m scenario when evaluated on the 8s vs. 8s scenario. The best performing policy in each category (trained with Marines and trained with Stalkers) per scenario is in bold.

the network, but did not see any change in performance. Since the size of the Stalkers is larger, it's possible that the size of the filters would need to be changed to learn to recognize useful information about larger groups of units. Another aspect of the training that may be causing poor performance is training time. We did these experiments with 300k steps since we saw no improvement in Marine scenarios over that amount of training. In informal testing after seeing these results, we saw no improvement when increasing training time to 600k steps and leaving everything else the same. We also tried to train on the Stalker scenarios without shield data in the state and reward, but had the same poor results. The last difference from training on Marine scenarios is that the Stalker scenarios use larger health ranges for their 3 categories of health since they have higher maximum health (80) than Marines (45). It is possible that training on Marine scenarios is easier because the model receives higher resolution health category information, which allows it to more easily learn relationships between the rewards it receives and the state.

Results for the models trained on Marine scenarios show that, while the scores for these Stalker scenarios are not as high as the Marine scenarios, the policies can indeed be transferred to units of different types. In particular, we can see that the models perform well on smaller sized scenarios, with scores tapering off for larger scenarios. We believe this is due to the fact that as more units enter the battlefield, the differences between those units such as size and damage type become more apparent, causing the policy to perform worse. One surprising result was that the policy trained on 16 vs. 16 Marines for 300k steps performed much worse than the other policies when evaluated on the Stalker scenarios. When we investigated by watching the policy play in real-time, we could see that the policy learns to attack targeting the ground very close to its own units. This has the effect of grouping the units very close together, which can be effective for the marine scenarios. When that policy is transferred to scenarios with other larger units, it sometimes targets its own units because of the size difference. Since training directly on Stalker scenarios may require more training steps or a larger network, both of which mean longer training time, these results show that training with some of the simplest units may be a good strategy for learning effective policies across a large range of unit types.

Results of models trained on Hydralisk scenarios and Marine scenarios and evaluated on Hydralisk scenarios are presented in Table 5.5. Similar to the results for models trained on Stalker scenarios, the models trained on Hydralisk scenarios do not perform well, and in fact have similar policies to the ones trained on Stalker scenarios. The Hydralisk scenario trained models also have similar performance to the NA scripted player in Table 5.2. These results are surprising and we have fewer ideas as to why the performance is not as high as on the Marine scenarios. Hydralisks don't have shields, so the state input size is the same as for the models trained on Marine scenarios. Like Stalkers, Hydralisks do have higher maximum health (90) than Marines (45) and thus have larger ranges for the health categories in the state input. However, Hydralisks have almost twice the DPS of Marines, and thus the ratio of health to DPS is very similar in the Hydralisk scenarios. Hydralisks are larger, like the Stalkers, supporting the possibility that unit size is may be affecting training performance.

Unit Type transfer from Marine scenarios is successful for Hydralisk scenarios. The best models trained on Marine scenarios perform have scores ranging from 0.575 in the 4h vs. 4h scenario to 0.349 in the 16h vs. 16h scenario. The model trained on the 8m vs. 8m scenario

Evaluation	Learned Policy - 300k Steps Trained on Marines				Learned Policy - 300k Steps Trained on Hydralisks			
Scenario	4m	8m	16m	32m	4h	8h	16h	32h
4h vs. 4h	0.543	**0.575**	0.059	0.450	**0.189**	0.151	0.156	0.116
8h vs. 8h	0.484	**0.529**	0.084	0.451	0.153	**0.209**	0.189	0.198
16h vs. 16h	0.271	0.346	0.078	**0.349**	0.073	0.188	**0.192**	0.190
32h vs. 32h	0.323	**0.505**	0.083	0.242	0.024	**0.106**	0.089	0.102

Table 5.5: Experiment scores ((wins + draws/2)/# episodes) of policies trained on Marine and Hydralisk scenarios, evaluated for 1k episodes in Hydralisk scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the 300k 4m column, 8h vs. 8h row gives the score of the model trained for 300k steps on the 4m vs. 4m scenario when evaluated on the 8h vs. 8h scenario. The best performing policy in each category (trained with Marines and trained with Hydralisks) per scenario is in bold.

scores 0.505 on the 32h vs. 32h scenario, which is the best result attained on any scenario with 32 units per side with a model trained for 300k steps. As with the Stalker results, these results show that training on one easier to learn scenario can be an effective way to see good results on other harder scenarios.

The Zergling scenarios are the only ones with melee units, so we expected that the best policies, and also training performance, would be different on them when compared to our other combat scenarios. Table 5.6 gives the results of models trained on both Marine and Zergling scenarios when evaluated on Zergling scenarios. It shows that performance of models trained on Zergling scenarios is on par with that of models trained on Marine scenarios evaluated on Marine scenarios. The model trained on 4z vs. 4z, however, only has a score of 0.413 on the 4z vs. 4z scenario, which perhaps indicated that this scenario is difficult to train in relative to the other Zergling and Marine scenarios. The models trained on the 8z vs. 8z and 16z vs. 16z scenarios have the best performance on most scenarios. Zergling scenarios with few units, especially the 4z vs. 4z scenario, end very quickly since the units are fast and have low maximum health. In this scenario in particular the advantage of the built-in AI opponent starting to move to attack immediately when the scenario begins may make the scenario hard to learn in. From our examination, the models trained on

Evaluation Scenario	Learned Policy - 300k Steps Trained on Marines 4m	8m	16m	32m	Learned Policy - 300k Steps Trained on Zerglings 4z	8z	16z	32z
4z vs. 4z	0.535	**0.586**	0.431	0.417	0.413	0.557	**0.670**	0.453
8z vs. 8z	0.533	**0.543**	0.365	0.406	0.322	0.579	**0.662**	0.360
16z vs. 16z	0.503	**0.528**	0.322	0.309	0.260	0.559	**0.578**	0.341
32z vs. 32z	0.447	**0.479**	0.158	0.214	0.456	**0.528**	0.420	0.447

Table 5.6: Experiment scores ((wins + draws/2)/# episodes) of policies trained on Marine and Zergling scenarios, evaluated for 1k episodes in Zergling scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the 300k 4m column, 8z vs. 8z row gives the score of the model trained for 300k steps on the 4m vs. 4m scenario when evaluated on the 8z vs. 8z scenario. The best performing policy in each category (trained with Marines and trained with Zerglings) per scenario is in bold.

Zergling scenarios have mainly learned to select all units and target the ground, similar to the models trained on Marine scenarios.

As with the Stalker and Hydralisk scenarios, Marine trained models perform well in general on the Zergling scenarios, achieving top scores ranging from 0.479 (32z) to 0.586 (4z) across the different Zergling scenarios. In this case, however, the models trained on the Zergling scenarios perform better. This result shows that it is possible to train a model on scenarios involving a single unit type in a complex game such as StarCraft II and use it for multiple scenarios having unit types with different properties.

5.5.6 Training on Multiple Scenarios Simultaneously

In addition to training on specific sizes of scenarios, we also experimented with training on a scenario which randomizes the number of units per episode. In the 4-32m vs. 4-32m scenario, the two sides start with an equal number of units in each episode, but the number of units per side is chosen randomly from between 4 and 32 units, inclusive. We expected that training on this scenario would produce a more robust policy that performs well in all the sizes trained, since experiences from both small and large scenarios would be in the replay memory at the same time, and whichever experiences offered the most to learn from

	Learned Policy - 300k Steps	
Evaluation	Trained on 4-32m vs. 4-32m	
Scenario	Avg.	Max.
4-32m vs. 4-32m	0.398	0.580
4m vs. 4m	0.525	0.774
8m vs. 8m	**0.553**	**0.821**
16m vs. 16m	0.443	0.639
32m vs. 32m	0.208	0.315

Table 5.7: Experiment scores ((wins + draws/2)/# episodes) of 10 models trained on the 4-32m vs. 4-32m scenario for 300k steps, evaluated for 1k episodes in the 4-32m, 4m, 8m, 16m, and 32m scenarios. The scores for the scenario with the highest win rate are in bold.

at any time according to their priority would be selected more often for training updates. We trained 10 models on this scenario for 300k steps each, and then evaluated the learned policies on the training scenario as well as each fixed-size Marine scenario. The average and maximum evaluation scores per scenario are presented in Table 5.7. The results show that training on this mixed-size scenario achieves very good scores on the smaller-sized scenarios, including the best score seen in the 8m vs. 8m scenario, but does not improve performance in the 32m vs. 32m scenario when compared to the results in Table 5.3.

We examined the learned policies visually and found that the best models trained on the 4-32m vs. 4-32m scenario tend to attack targeting the ground near the their own units. It is notable that they have not learned to focus fire on enemy units, as the AWN scripted policy and some other policies trained only on smaller scenarios do (and which does not perform well in the scenarios with 16 and 32 units), implying that training on this mixed-size scenario caused the models to learn policies that do not perform badly in any of the included sizes.

5.5.7 Curriculum Transfer Learning

We tested two different curriculums for training models, in which we start by training in one easier scenario, and then train the model on one or more harder scenarios in sequence. In the first experiment, we trained models in the 8m vs. 8m scenario, and then trained those same models for more time on the 8s vs. 8s scenario. Results from Section 5.5.5 show that it is harder to learn good policies for the Stalker scenarios than for the Marine scenarios, whether by directly training in the Stalker scenarios or by training in the Marine scenarios and applying those policies to Stalker scenarios. In the second experiment, we trained models in progressively larger Marine scenarios, expecting to see better performance in the larger scenarios. Results from Sections 5.4 and 5.5.4 show that the 32m vs. 32m scenario is the hardest of our Marine scenarios.

5.5.7.1 Unit Type Curriculum Transfer Learning

Results from the unit type CTL experiment are presented in Table 5.8. We trained 3 models on the 8m vs. 8m scenario, and present the scores those models achieved on both the 8m vs. 8m scenario and the 8s vs. 8s scenario. To continue training with an existing model, we make a copy of it and repeat the training process with a new replay memory starting from random actions, as described in Section 3.2.3. We tried using the same exploration schedule, starting with the exploration parameter $\epsilon = 1$ and exponentially annealing to 0.05 over the first 80% of training steps. We also tried training the new models by starting with $\epsilon = 0.5$, since the model is starting in an already trained state and we anticipated that purely random exploration would not be as useful. We applied the two versions of additional training on the 8s vs. 8s scenario to all three models, and present the scores of those final models on the 8s vs. 8s scenario only. Since the curriculum includes a Stalker scenario, the unit_shields feature layers are in the state input, including during training on the Marine scenario. This is the only time a model was trained on a Marine scenario with shield input data, and the values of those inputs would have all been 0 during the first 300k steps of training.

The results show that additional training on the 8s vs. 8s scenario leads to better performance in it. Of the original three models, models 2 and 3 have the best performance on the training scenario, 8m vs. 8m. All three models perform about the same, and not very well, in the 8s vs. 8s scenario. By watching the policies in real-time on the 8m vs. 8m scenario, we

Training Curriculum	Evaluation Scenario	Run # 1	2	3
8m vs. 8m 300k Steps	8m vs. 8m	0.116	0.676	**0.682**
8m vs. 8m 300k Steps	8s vs. 8s	0.161	**0.180**	0.159
8m vs. 8m 300k Steps, 8s vs. 8s 300k Steps (Starting $\epsilon = 1$)	8s vs. 8s	0.340	**0.393**	0.196
8m vs. 8m 300k Steps, 8s vs. 8s 300k Steps (Starting $\epsilon = 0.5$)	8s vs. 8s	0.290	**0.389**	0.191

Table 5.8: Experiment scores ((wins + draws/2)/# episodes) of three models trained for 300k steps on the 8m vs. 8m scenario, then trained for an additional 300k steps on the 8s vs. 8s scenario following one of two exploration schedules. All models were evaluated for 1k episodes on the specified scenarios. The score of the best performing model in each trainingevaluation combination is in bold.

see that model 1 learned to mostly attack (0,0), sometimes attack the ground near its own units or the enemies, and sometimes attacks its own units, explaining its poor performance. Models 2 and 3 learned to target near enemies, and model 2 learned to focus fire on individual enemies when the number of enemies is low. Those policies behaved similarly in the Stalker scenario. Surprisingly, starting the second training with $\epsilon = 1$ led to better performing policies for all three original models. Also surprisingly, the best performing Marine policy, run # 3, made for the worst Stalker policy after additional training. That model also had the worst Stalker policy after only the initial training, so performance in the target scenario may be a better indicator of performance after further training. Looking at the actual policies of the models trained for the full 600k steps, it is difficult to distinguish between the models trained with the two different starting values of ϵ, despite the large differences in score. The models based on run # 1 kept a similar policy, but without targeting its own units. The models based on run # 2 learned to do more focus firing than the base model, while the models based on run # 3 learned to focus fire even more than the # 2 models.

Overall these results suggest that the method of picking the top performing model trained on one scenario may not always be the best method to find good policies in other scenarios.

Additionally, the best policy for these scenarios is likely a mix of tactics that might include tactics our models rarely learn, such as selecting small groups of units at different times.

5.5.7.2 Battle Size Curriculum Transfer Learning

To test CTL with increasing battle size we trained models on the 4m vs. 4m model and then continued to train those models for more steps on the 8, 16, and 32 Marine size scenarios. For these experiments we trained models for 100k steps on each scenario because of the large amount of training required. We trained 5 models on the 4m vs. 4m scenario initially, and then tried 3 different ways to choose which models to copy when doing further training. First, we trained each 4m model for a further 100k steps on the 8m vs. 8m scenario, followed by 100k steps on the 16m vs. 16m scenario, and finally 100k steps on the 32m vs. 32m model. Second, we chose the best performing of the 4m models (evaluated on the 4m vs. 4m scenario) and created 5 separate 8m vs. 8m training runs from that one model. Subsequently we chose one 8m model and then one 16m model in the same way to train on the 16m vs. 16m and 32m vs. 32m scenarios, respectively. The third way we chose models to train on was the same as the second method, but using the score of models on the *next* scenario in the curriculum to choose the model to train from for the next scenario. Results from Section 5.5.7.1 indicate that performance of a model on the next scenario in the curriculum may be a good indicator of performance after training in that scenario. All training was done with exploration parameter $\epsilon = 1$ at the start of training, since that produced the best results in Section 5.5.7.1. Results for these three methods are presented in Table 5.9.

The results of these experiments don't suggest that any of the methods we tested are particularly effective for CTL up to the scenarios with 32 units. Using the first method, performance declines as scenario size increases, similar to models trained only on those scenarios. For the second and third methods the 4m vs. 4m trained model chosen to copy is the same one, since it performed best on both the 4m vs. 4m and 8m vs. 8m scenarios. The second and third methods each have one stage of training (32m vs. 32m and 16m vs.

	Scenario Evaluation Score							
Training	4m vs. 4m		8m vs. 8m		16m vs. 16m		32m vs. 32m	
Curriculum (Steps)	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
4m vs. 4m (100k)	0.330	0.630	0.093	0.147	-	-	-	-
Each model trained on next scenario								
4m, 8m (100k each)	-	-	0.348	0.679	-	-	-	-
4m, 8m, 16m (100k each)	-	-	-	-	0.313	0.569	-	-
4m, 8m, 16m, 32m (100k each)	-	-	-	-	-	-	0.169	0.240
Model with best score on current scenario is trained on next scenario								
4m, 8m (100k each)	-	-	0.673	0.708	-	-	-	-
4m, 8m, 16m (100k each)	-	-	-	-	0.313	0.321	-	-
4m, 8m, 16m, 32m (100k each)	-	-	-	-	-	-	0.041	0.045
Model with best score on next scenario is trained on next scenario								
4m, 8m (100k each)	-	-	0.673	0.708	0.562	0.596	-	-
4m, 8m, 16m (100k each)	-	-	-	-	0.032	0.042	0.108	0.276
4m, 8m, 16m, 32m (100k each)	-	-	-	-	-	-	0.290	0.317

Table 5.9: Experiment scores ($(\text{wins} + \text{draws}/2)/\#$ episodes) of five models trained for 100k steps on the 4m vs. 4m scenario, then trained for additional periods of 100k steps on progressively larger-sized Marine scenarios following three different methods for selecting models to continue training. All models were evaluated for 1k episodes on the specified scenarios. Values of "-" indicate combinations that were not a part of any curriculum.

16m, respectively) where performance was very low for all five runs. This suggests that choosing only one model to continue to the next stage of training risks having a model that has learned some poor behaviour that might not be hurting it on the smaller scenarios but which is hard to train out of. The first method benefits from maintaining a diversity of models at each step. The third method did produce the best result on the 32m vs. 32m scenario, so it's possible that a mix of the first and third method, in which a few of the top performing models would be copied for further training, would work well.

Chapter 6

Conclusion

In this chapter we summarize the contributions made in this thesis and present ideas for future research in this area.

6.1 Summary

In this thesis we presented our implementation of component-action DQN for learning to play combat scenarios in the complex RTS game StarCraft II. In Section 4.3.1 we described the challenges posed by the human-like action interface in the SC2LE, and our solutions for handling that action interface using component-action DQN. Actions in SC2LE have a function part and multiple parameters which vary based on the function choice, rather than a single action choice as in most other RL game environments. We used component masking when calculating training loss and tested multiple ways to calculate training loss. In section 5.3 we showed that comparing primary network component action values to an averaged target network next-state value resulted in the best performance, on average.

In our experiments we showed that with short training times and a relatively easy to implement RL system, our RL players could defeat the built-in AI most of the time in basic combat scenarios with smaller unit counts. In Section 5.5.4 we successfully demonstrated transfer learning between combat scenarios of different sizes: that policies can be learned in a scenario of a given size, and then applied to scenarios of different sizes with comparable results. We also demonstrated transfer learning between different unit types in Section 5.5.5, with policies learned in scenarios with Marine unit battles being successfully applied to battles with Stalker, Hydralisk, and Zergling units. In some cases the policies learned by training on the Marine scenarios performed better on the scenarios with other unit types than the policies trained on those unit types.

We also explored several other ways to attain higher performance on the more difficult scenarios (certain unit types and larger battle sizes). In section 5.5.6 we showed that training on a range of unit sizes at once (with random episode starting army sizes) was effective for learning policies that performed well on all sizes on average, and in particular for small sized scenarios, but was not a good solution for learning better policies for large scenarios. In section 5.5.7 we tested CTL with curriculums that either increased the unit count or changed the unit type in separate training runs. We were not able to improve performance on the most difficult scenarios using the CTL techniques that we tried, but we identified some techniques that do not work well. We found that decreasing the starting exploration rate in later training runs did not improve performance. We also found that both continuing training from models without evaluating them and continuing only the single best performing model were not successful strategies.

6.2 Conclusion

The results of this thesis are promising and show that training on small scenarios for short training times gives models that are applicable in a larger set of scenarios. We consider each of our research questions:

1. *Can we train a model on one StarCraft II combat scenario and then apply it to another scenario, which differs in some non-trivial way such as size or unit type, and achieve similar performance.*

As mentioned in Section 6.1, this technique can indeed lead to similar and sometimes better performance in scenarios which the model has not been trained on.

2. *How suitable is the DQN RL algorithm to this type of problem when modified to use an action space similar to human-control.*

   While we had success in transferring learning to small scenarios and scenarios with different unit types, we were not successful in training models with higher than 0.5 score in the largest scenarios with 32 units. Longer training times and larger state representations did not result in better performance. This indicates that the DQN algorithm may not be well suited to this problem. Alternatively, other enhancements to the network structure may be needed to achieve better performance.

We believe that these results show promise for the future of RL in RTS games, by allowing us to train policies in smaller, less complex scenarios, and then apply those policies to different areas of the game, reducing the need for longer training times and much larger networks, like those used to train AlphaStar [49].

6.3 Contribution

This thesis makes several contributions to research in RTS AI. We explored different options for using component actions with DQN, including how to calculate training loss when masking actions due to some action components only being used for some actions. In Section 5.3 we present results showing the performance of different training loss formulas. We also showed that trained models could maintain performance in scenarios other than the ones they are trained on. For example, in Section 5.5.5 we show that the models trained on Marine scenarios outperform the models trained on Hydralisk scenarios when evaluated on Hydralisk scenarios. Several transfer learning methods we tried were not successful in improving performance, as shown in Section 5.5.7. These results help to indicate what some future areas of research might be, which are discussed in Section 6.4.

6.4 Future Work

We believe that the results presented in this thesis indicate several directions for future work. In our experiments, we found one set of hyperparameters and network structure which worked well for some basic combat scenarios, and then applied those to some other scenarios and tried different training techniques such as CTL. These training parameters were not sufficient to perform at high levels in some of the scenarios we tested, and therefore further experiments with different network structures, longer training times, and larger replay memory size would be useful for determining the limiting factors in learning policies for these more complex scenarios. Results with CTL suggest that other protocols for choosing models to continue training from, such as training multiple new models from each of several best performing models in a previous part of the curriculum, may perform better.

Evaluating the suitability of the RL algorithm and the component-action method generally could be improved by testing in scenarios that require using more action types to achieve high scores. Additionally, testing in scenarios with different opponents, for example scripted players with controlled behaviour rather than the built-in AI, could also reveal more insights about the strengths and weaknesses of the trained models. Additionally, further experiments in an overall less complex RTS environment such as microRTS, but with scenarios involving multiple units, bases, etc. could be used to better test RL algorithms and network structures.

We chose to use DQN for doing experiments in the SC2LE domain because it has been successful in other domains, and it works well for discrete action spaces. However, other more recent RL algorithms such as A2C have outperformed DQN in benchmark domains such as the ALE. In particular with component-action RL algorithms the exploration technique used may be important, since exploration is happening in multiple action components at once. Policy gradient algorithms such as A2C are on-policy and explore naturally through their stochastic policies, as opposed to epsilon-greedy exploration in DQN. Future work for this research can include implementing a similar network architecture and action component system using policy gradient RL methods to compare them to component-action DQN. We believe that applying component-action systems to these other RL algorithms is a promising avenue for future research.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov 2017. `doi:10.1109/MSP.2017.2743240`.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[4] Michal Čertickỳ, David Churchill, Kyung-Joong Kim, Martin Čertickỳ, and Richard Kelly. StarCraft AI competitions, bots, and tournament manager software. *IEEE Transactions on Games*, 11(3):227–237, 2018.

[5] David Churchill. *Heuristic Search Techniques for Real-Time Strategy Games.* PhD thesis, University of Alberta, 2016.

[6] David Churchill, Zeming Lin, and Gabriel Synnaeve. An analysis of model-based heuristic search techniques for StarCraft combat scenarios. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2017.

[7] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontañnón, and Michal Čertický. StarCraft bots and competitions. In Newton Lee, editor, *Encyclopedia of Computer Graphics and Games*, pages 1–18. Springer International Publishing, Cham, 2016. `doi:10.1007/978-3-319-08234-9_18-1`.

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI baselines. `https://github.com/openai/baselines`, 2017.

[9] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *International Conference on Learning Representations*, 2018. URL: `https://openreview.net/forum?id=rywHCPkAW`.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[12] Adam Heinermann. Broodwar API. `https://github.com/bwapi/bwapi`, 2013.

[13] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/11796`.

[14] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning: Lecture 6a: Overview of mini-batch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 2020-11-17.

[15] Yue Hu, Juntao Li, Xi Li, Gang Pan, and Mingliang Xu. Knowledge-guided agent-tactic-aware learning for StarCraft micromanagement. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1471–1477. International Joint Conferences on Artificial Intelligence Organization, 7 2018. `doi:10.24963/ijcai.2018/204`.

[16] Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in MicroRTS. *arXiv preprint arXiv:1910.12134*, 2019.

[17] tf.keras.losses.huber. `https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/keras/losses/Huber`. Accessed: 2020-11-18.

[18] N. Justesen, P. Bontrager, J. Togelius, and S. Risi. Deep learning for video game playing. *IEEE Transactions on Games*, pages 1–1, 2019. `doi:10.1109/TG.2019.2896986`.

[19] N. Justesen and S. Risi. Learning macromanagement in StarCraft from replays using deep learning. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 162–169, Aug 2017. `doi:10.1109/CIG.2017.8080430`.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: `http://arxiv.org/abs/1412.6980`.

[21] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a TensorFlow library for applied reinforcement learning. Web page, 2017. URL: `https://github.com/tensorforce/tensorforce`.

[22] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, pages 2140–2146, 2017. URL: `https://arxiv.org/pdf/1609.05521.pdf`.

[23] Liquipidia. Unit statistics (Legacy of the Void). `https://liquipedia.net/starcraft2/Unit_Statistics_(Legacy_of_the_Void)`. Accessed: 2020-09-28.

[24] 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 Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: `http://proceedings.mlr.press/v48/mniha16.html`.

[25] 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*, 2013.

[26] 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. URL: `http://files.davidqiu.com/research/nature14236.pdf`.

[27] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'13, page 58–64. AAAI Press, 2013.

[28] Santiago Ontañón, Nicolas A. Barriga, Cleyton R. Silva, Rubens O. Moraes, and Levi H. S. Lelis. The first microRTS artificial intelligence competition. *AI Magazine*, 39(1):75–83, Mar. 2018. URL: `https://www.aaai.org/ojs/index.php/aimagazine/article/view/2777`, `doi:10.1609/aimag.v39i1.2777`.

[29] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. RTS AI problems and techniques. In Newton Lee, editor, *Encyclopedia of Computer Graphics and Games*, pages 1–12. Springer International Publishing, Cham, 2015. `doi:10.1007/978-3-319-08234-9_17-1`.

[30] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. On reinforcement learning for full-length game of StarCraft. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, volume 33, pages 4691–4698, 2019.

[31] A. L. Samuel. Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[32] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. The StarCraft multi-agent challenge. *CoRR*, abs/1902.04043, 2019.

[33] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21, Mar. 1996. URL: `https://www.aaai.org/ojs/index.php/aimagazine/article/view/1208`, `doi:10.1609/aimag.v17i1.1208`.

[34] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL: `http://arxiv.org/abs/1511.05952`.

[35] 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. `arXiv:https://doi.org/10.1080/14786445008521796`, `doi:10.1080/14786445008521796`.

[36] A. Shantia, E. Begue, and M. Wiering. Connectionist reinforcement learning for intelligent unit micro management in StarCraft. In *The 2011 International Joint Conference on Neural Networks*, pages 1794–1801, July 2011. `doi:10.1109/IJCNN.2011.6033442`.

[37] K. Shao, Y. Zhu, and D. Zhao. StarCraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 3(1):73–84, Feb 2019. `doi:10.1109/TETCI.2018.2823329`.

[38] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[39] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354—-359, 2017.

[40] M. Stanescu, N. A. Barriga, A. Hess, and M. Buro. Evaluating real-time strategy game states using convolutional neural networks. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7, Sep. 2016. `doi:10.1109/CIG.2016.7860439`.

[41] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, and Tong Zhang. TStarBots: Defeating the cheating level builtin AI in StarCraft II in the full game. *arXiv preprint arXiv:1809.07193*, 2018.

[42] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press Cambridge, 2nd edition, 2018.

[43] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. TorchCraft: A library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016. URL: `https://arxiv.org/abs/1611.00625`.

[44] Z. Tang, D. Zhao, Y. Zhu, and P. Guo. Reinforcement learning for build-order production in StarCraft II. In *2018 Eighth International Conference on Information Science and Technology (ICIST)*, pages 153–158, June 2018. `doi:10.1109/ICIST.2018.8426160`.

[45] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. Action branching architectures for deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 4131–4138, 2018.

[46] A. Uriarte and S. Ontañón. Combat models for RTS games. *IEEE Transactions on Games*, 10(1):29–41, March 2018. `doi:10.1109/TCIAIG.2017.2669895`.

[47] Nicolas Usunier, Gabriel Synnaeve, Zeming Lin, and Soumith Chintala. Episodic exploration for deep deterministic policies: An application to StarCraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*, 2016. URL: `https://arxiv.org/abs/1609.02993`.

[48] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[49] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the real-time strategy game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[50] 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.

[51] 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. URL: `https://arxiv.org/abs/1708.04782`.

[52] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of Machine Learning Research*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: `http://proceedings.mlr.press/v48/wangf16.html`.