Evaluation of Deep Learning Methods for Creating Synthetic Actors

Babak Toghiani-Rizi
Abstract

Evaluation of Deep Learning Methods for Creating Synthetic Actors

Babak Toghiani-Rizi

Recent advancements in hardware, techniques and data availability have resulted in major advancements within the field of Machine Learning and specifically in a subset of modeling techniques referred to as Deep Learning. Virtual simulations are common tools of support in training and decision-making within the military. These simulations can be populated with synthetic actors, often controlled through manually implemented behaviors, developed in a streamlined process by domain doctrines and programmers. This process is often time inefficient, expensive and error prone, potentially resulting in actors unrealistically superior or inferior to human players.

This thesis evaluates alternative methods of developing the behavior of synthetic actors through state-of-the-art Deep Learning methods. Through a few selected Deep Reinforcement Learning algorithms, the actors are trained in four different light weight simulations with objectives like those that could be encountered in a military simulation.

The results show that the actors trained with Deep Learning techniques can learn how to perform simple as well as more complex tasks by learning a behavior that could be difficult to manually program. The results also show the same algorithm can be used to train several totally different types of behavior, thus demonstrating the robustness of these methods.

This thesis finally concludes that Deep Learning techniques have, given the right tools, a good potential as alternative methods of training the behavior of synthetic actors, and to potentially replace the current methods in the future.
Populärvetenskaplig sammanfattning

Under den senaste tiden har en teknisk utveckling i kombination med tillgänglighet av information påverkat en rad olika områden i samhället. Ett av dessa områden är artificiell intelligens, men också en av metoderna för att skapa artificiell intelligens - maskininlärning - där man skapar förutsättningarna för en dator eller ett program själv tränas till att uppnå artificiell intelligens.

Inom det militära har det länge varit vanligt att nyttja virtuella medel, så som simuleringar, för att underlättar genomförandet av tränings, övningar och beslutfattande. Dessa simuleringar populeras ibland av intelligenta aktörer med ett manuellt förprogrammerat beteendemönster, ofta på formen av ett beteendeträd. Metoden för att framställa dessa är dock kostsamt och resurskrävande, och det den fina balansgången mellan att beteendet ska uppfattas som övermänskt kontra ointelligent som gör att det också blir väldigt svårt att skapa aktörer som till fullo är realistiska.

Den senaste tidens utveckling inom maskininlärning har drastiskt förenklat möjligheten att skapa intelligenta system som kan utföra mer komplexa uppgifter, ofta med hjälp av djupinlärning av djupa neurala nätverk. Ofta har dessa tekniker nyttjats för klassificering och regression, men senaste tidens utveckling har också kommit att underlättat möjligheten för att skapa intelligenta aktörer i simuleringar. Genom att låta aktörer agera i en miljö med realistiska förutsättningar kan djupinlärningstekniker nyttjas för att aktören själv ska skapa ett realistiskt beteende, ofta med små resurser och på mindre tid än manuellt skapande.

Denna rapport syftar till att identifiera några av de absolut främsta av dessa metoder, samt att utvärdera deras möjlighet för att skapa beteenden för scenarion som skulle kunna förekomma i militära simuleringar.
Acknowledgments

This thesis was supported by the Swedish Defence Research Agency (FOI) project *Synthetic Actors*, a project funded by the R&D program of the Swedish Armed Forces.

I would like to begin this thesis with thanking my supervisors. Thank you Dr Linus Gissén (previously the Swedish Defence Research Agency - FOI, now Frostbite Labs - DICE, EA) and Dr Linus Luotsinen (Swedish Defence Research Agency - FOI) for believing in me and giving me the opportunity do a deep dive into the research area that really lights my fire. You are the best!

Thank you, Dr Michael Ashcroft (Uppsala University) who reviewed this thesis. Thank you for your support and for the effort you put into my path of putting this together and for arranging the weekly meetings every Friday.

Thank you, fellow Friday-meeting peers for your support, ideas, questions and answers. It really was of great help to have you around to discuss all things ML.

Thank you, Olle Gällmo (Uppsala University) for your course on Machine Learning. What started out as an initial spark of interest and curiosity for Machine Learning has eventually bloomed out as a passion of mine, and this course turned out to be crucial for what I do today.

I would like to dedicate this thesis to my friends and more importantly my family. Thank you all for your immense support and encouragement. You made this possible, this is for you.

*Babak Toghiani-Rizi, 2017*
# Contents

List of Figures 6

List of Acronyms 8

1 Introduction 11
   1.1 Background ........................................ 12
   1.2 Purpose ........................................... 13
      1.2.1 Problem Statement ............................ 15
   1.3 Delimitations .................................... 15

2 Related Work 16
   2.1 Deep Learning in Games .......................... 17
   2.2 Machine Learning Methods in Military Simulations 18

3 Theory 19
   3.1 Reinforcement Learning ............................ 20
      3.1.1 Value Iteration .............................. 21
      3.1.2 Actor-Critic .................................. 22
      3.1.3 Q-Learning ................................... 23
   3.2 Artificial Neural Networks ....................... 24
      3.2.1 Training an Artificial Neural Network .......... 26
      3.2.2 Combining Artificial Neural Networks and Reinforce-
         ment Learning .................................. 27
   3.3 Convolutional Neural Networks .................... 27
      3.3.1 Convolutional Layer .......................... 28
      3.3.2 Pooling Layer ................................ 29
      3.3.3 Fully Connected Layer ......................... 30
   3.4 Recurrent Neural Networks ........................ 30
      3.4.1 Long Short-Term Memory ....................... 32
   3.5 Deep Learning .................................... 34
      3.5.1 Deep Reinforcement Learning Methods .......... 34
         3.5.1.1 Deep Q-Learning .......................... 35
         3.5.1.2 Dueling Deep Q-Learning ................ 36
      3.5.2 Asynchronous Deep Reinforcement Learning Methods 38
List of Figures

3.1 The Agent-Environment Interaction model. . . . . . . . . . . 21
3.2 The Agent Behavior model. . . . . . . . . . . . . . . . . . . 21
3.3 The Actor-Critic Interaction model. . . . . . . . . . . . . . 23
3.4 An Artificial Neuron with 3 inputs. . . . . . . . . . . . . . . 25
3.5 An Artificial Neural Network architecture example with five
inputs, three units in the hidden layer and one output node. . 26
3.6 An illustrated example of a Convolutional Neural Network
and how it operates with a volume of neurons in three dimen-
sions. The leftmost layer represents the input with a width,
height and depth, followed by two layers of three dimen-
sional convolutional operations and ultimately a fully con-
nected layer that connects to an output. . . . . . . . . . . . . 28
3.7 An illustrated example of a horizontal slide in convolution on
a 7x7 input using a 3x3 filter map and stride length 1. . . . . 28
3.8 An illustrated example of a horizontal slide in convolution on
a 7x7 input using a 3x3 filter map and stride length 2. . . . . 29
3.9 Examples of an input image (top) run through various fil-
ters (middle) for edge detection, and their respective output
(bottom). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.10 An illustration showing how pooling down samples the width
and height of a volume while keeping spatial information and
the input volume. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.11 Examples of two different pooling techniques. . . . . . . . . 30
3.12 Recurrent Neural Network model, folded (left) and unfolded
(right) with sequence of length $n$. . . . . . . . . . . . . . . . . 31
3.13 The structure and operations of a Recurrent Neural Network
unit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.14 The structure and operations of a Long Short-Term Memory
cell. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.15 Hierarchical features of facial recognition, showing how the
features learned to go from edges, facial features and ulti-
mately to full faces with the depth of the network (left to
right). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.16 A Deep Q-Network Architecture example with outputs corresponding four different actions.

3.17 A Dueling Deep Q-Network Architecture example with outputs corresponding four different actions.

3.18 Example of an A3C Network Architecture with a Feed-Forward Neural Network.

3.19 Example of an A3C Network Architecture with a Neural Network with LSTM Cells.

4.1 An illustrated sequence of the first simulation where agent is rewarded by collecting items.

4.2 An illustrated sequence of the second simulation where agent is rewarded by collecting items in an area with obstacles.

4.3 An illustrated sequence of the third simulation where agent is rewarded by guarding a moving target in cooperation with a programmed actor. The reward is based on the guarded area.

4.4 An illustrated sequence of the fourth simulation where agent is rewarded by advancing towards goal. The reward is also based on advancing within the guarded area and guarding an area, so the programmed actor can advance.

5.1 The average reward/episode during the training of the regular agent models of Experiment 1.

5.2 The average reward/episode during the training of the asynchronous agent models of Experiment 1.

5.3 The performance of each model in Experiment 1.

5.4 The average reward/episode during the training of the regular agent models of Experiment 2.

5.5 The average reward/episode during the training of the asynchronous agent models of Experiment 2.

5.6 The performance of each model in Experiment 2.

5.7 The average reward/episode during the training of the regular agent models of Experiment 3A.

5.8 The average reward/episode during the training of the asynchronous agent models of Experiment 3A.

5.9 The performance of each model in Experiment 3A.

5.10 The average reward/episode during the training of the regular agent models of Experiment 3B.

5.11 The average reward/episode during the training of the asynchronous agent models of Experiment 3B.

5.12 The performance of each model in Experiment 3B.

5.13 The average reward/episode during the training of the regular agent models of Experiment 4.
5.14 The average reward/episode during the training of the asynchronous agent models of Experiment 4. . . . . . . . . . 62
5.15 The performance of each model in Experiment 4. . . . . . . . . 62

6.1 Trace plot of the best performing agent (A3C-LSTM). . . . . 65
6.2 Trace plot of the best performing agent (A3C-LSTM). . . . . 66
6.3 Trace plot of the best performing agent of 3B (A3C-LSTM). 68
6.4 The average reward/episode during the training of the asynchronous agent models in the extended Experiment 4. . . . . 70
6.5 Performance showing the improvement of the models between maximum training step set to ($T_{max}$) 50 million training steps and 80 million training steps. . . . . . . . . . . . . . . . . . . . 70
6.6 Trace plot of the best performing agent (A3C-LSTM). . . . . 71
Acronyms

**A3C**  Asynchronous Advantage Actor-Critic. 38, 39, 64, 77

**A3C-FF**  A3C with a Feed Forward Network. 42, 44, 45, 64, 65, 69, 71

**A3C-LSTM**  A3C with an LSTM Network. 42, 44, 45, 64–67, 69–71

**AA**  Asynchronous Agent. 42, 43, 54, 56, 58, 59, 61, 64, 65, 69, 71, 76

**ADRL**  Asynchronous Deep Reinforcement Learning. 17, 38, 77

**AI**  Artificial Intelligence. 12, 13

**AN**  Artificial Neuron. 24, 25

**ANN**  Artificial Neural Network. 24–27, 30, 34, 39, 73

**BO**  Bounding Overwatch. 50

**CGFs**  Computer Generated Forces. 14, 18, 42

**CNN**  Convolutional Neural Network. 27, 30, 34, 37, 38, 74, 76

**DDQL**  Dueling Deep Q-Learning. 36, 38

**DDQN**  Dueling Deep Q-Networks. 37, 38, 42, 44, 45, 64

**DL**  Deep Learning. 12, 13, 17, 18, 34, 42, 73, 76, 77

**DNN**  Deep Neural Network. 34, 38

**DQL**  Deep Q-Learning. 17, 35, 36, 38, 42

**DQN**  Deep Q-Network. 17, 35–37, 42, 44, 45, 64

**DRL**  Deep Reinforcement Learning. 17, 18, 34, 35, 39, 42, 77

**GPU**  Graphical Processing Unit. 13

**LSTM**  Long Short-Term Memory. 32–34, 39, 67
MDP  Markov Decision Process. 20, 21, 27
MGD  Mini-batch Gradient Descent. 27
ML  Machine Learning. 12
RA  Regular Agent. 42, 43, 54, 56, 58, 59, 61, 64, 65, 69, 71, 76
RL  Reinforcement Learning. 17, 20, 23, 27, 34, 38
RNN  Recurrent Neural Network. 17, 31, 32, 34
SGD  Stochastic Gradient Descent. 27
VBS3  Virtual Battlespace 3. 13, 14, 76–78
Chapter 1

Introduction

This chapter serves as the background of this thesis by giving a brief summary of the history and evolution of Artificial Intelligence. Further, the current situation with the presence of Artificial Intelligence within military simulations used by the Swedish Armed Forces will be described, as well as how this thesis aims to evaluate a new frontier of techniques for developing intelligent behavior.
1.1 Background

For decades, the domain of Artificial Intelligence (AI) has been a subject that has been brought up in a wide spectrum of areas, ranging from research, science, philosophy and even all the way to literature and entertainment.

Alan Turing, the famous mathematician who greatly contributed to theoretical Computer Science and AI [1] named a test famously known as the Turing Test - a test measuring the intelligence of an artificial entity [2]. According to the Turing Test, an artificial entity that could exhibit intelligent behavior equivalent to or indistinguishable from that of a human would determine actual intelligence, a measurement that has been widely questioned and discussed since [3][4].

One of the early milestone achievements in AI took place in the early 1950’s, when the University of Cambridge-built computer EDSAC, later referred to as OXO, could play against and beat a human player in the game Tic-Tac-Toe [5]. This achievement extended the focus toward a more complex problem - such as a computer being able to play chess. Chess, it was argued, had a far larger state space and a specific set of rules that would increase the complexity even further [6] and turned out to be a challenge for many years to come. In 1996 IBM’s computer Deep Blue managed to beat the at that time reigning Chess master Garri Kasparov [7][8]. Again, in 2011, IBM reached another milestone when their computer Watson managed beat the two champions in Jeopardy - the reversed quiz where the players are given answer and have to find the appropriate questions for it [9].

Up until this point in time, the common method of creating AI was based on a constructed search tree over all possible states and actions, where increased complexity in the problems would lead to an exponential increase in the search tree [10].

The last decade’s advancements in computational power and the increased available data resources has sparked a renewed interest in the area in which computers, or machines, are trained to achieve some level of AI, referred to as Machine Learning (ML). Rather than using a traditional search tree, ML algorithms use complex models and parameters that can be used to perform a wide range of tasks such as compressing data, classifying objects, matching or completing patterns, detecting anomalies or to control - just to name a few. Along with the technological advancements and the increased availability of data, these models can achieve even better generalization, leading to increased robustness and higher accuracy. Therefore, the field of ML is already making an enormous impact in industry and large areas of research.

Deep Learning (DL) refers to a subset of methods within ML that utilize wider, deeper and even more complex models. These methods are able to automatically infer features from data rather than to rely on manually selected features. Theoretically and conceptually, DL originates from the mid
1960’s [11], but the computational limitations had been its catch. The number of mathematical computations in DL methods, the unstable algorithms and the amount of data it required made it too impractical and expensive to use for experiments - as it could take months to train a model if it even could learn successfully.

Along with the increased amount of available data, recent advancements in utilizing the Graphical Processing Unit (GPU) in a computer to distribute a large amount of smaller computations, such as matrix operations, has greatly contributed to pushing the field of DL forward, potentially shortening the training time of a model from months to hours [12].

One of the most recent major in advancements took place in early 2016, when Google DeepMind’s computer, AlphaGo, beat a human champion on the Chinese board game in four out of five matches [13]. Go is a significantly more complex game than Chess. With a total of $3^{361}$ possible game states (more than all the atoms of the universe), Go is not just an extremely demanding problem to solve only using search tree, but also requires more long-term tactics in order to play successfully against other skilled players [14]. The computer was trained using advanced DL methods, using both data from previous games to train it to a professional level, but also trained by playing against itself - allowing it to excel beyond human-level performance [15][16].

Along with many other recent advancements in AI, this event served as a symbol of the new era of intelligent systems - with DL in its frontier [17].

1.2 Purpose

Simulated tactical training in the military and simulation is often resource intensive and hard to manage, especially since it often requires a very specific set of actors involved in an environment that is not always close at hand. Therefore, it is common within the military to execute this type of virtual training, allowing more actors to be involved in an environment or setting that can be fully tailored according to the requirements beforehand.

Today, the Swedish Armed Forces (Försvarsmakten) use the tool Virtual Battlespace 3 (VBS3) to simulate ground combat to train and educate in decision-making and tactics. VBS3 provides a rich and realistic environment which users can control or view single military entities as well as large groups of military units in both 2D (birds eye-view, viewed from above) and 3D (viewed in first or third person). The military entities or units normally act on given command through input or scripted controls but can also act autonomously and make decisions on their own, based on manually implemented behavior models.

The current process of developing a single military entity’s behavior model requires a doctrine or domain expert within the respective field, who
has to create and describe a behavior scheme covering enough situations, states, parameters and actions to produce a good, realistic and general behavior model. This behavior then serves as specification to a developer, who then manually implements the behavior into a program that the agent then can execute upon. Unfortunately, this process is a costly and often error prone method that is time inefficient. A good, realistic behavior of the military entity is entirely dependent on the fact that the behavior-scheme is sufficiently complete and that the developer managed to implement it well enough so that the implementation has (preferably) no bugs. In reality, this is seldom true and there is a growing need for non-predictable and adaptive agents to improve the quality of virtual simulations.

The Swedish Defence Research Agency (Totalförsvarets Forskningsinstitut, FOI) is currently researching alternative approaches for creating Synthetic Actors. By examining various methods where behavior models are trained through data rather than implemented by hand. The purpose is to generate artificially intelligent Computer Generated Forces (CGFs) representing autonomous or partially autonomous military units such as pilots, soldiers, vehicles, but also aggregated military joints.

As a part that project, this thesis aims to explore how the CGFs can be trained by utilizing the most recent advancements within Deep Learning, and to evaluate the performance through a number of experiments with simulated objectives. The objectives represent down scaled versions of different tactical maneuvers, resembling objectives seen in the military setting and situations in VBS3 [18].
1.2.1 Problem Statement

- Is it possible to train artificial agents to perform well in different military situations?
- How do different variations of Deep Learning algorithms impact the model performance and training time?
- Can we achieve complex behavior by training an agent with Deep Learning, such as a specific tactic or way of taking actions, which would have been hard to do if the agent was scripted or implemented by hand?
- Could it be more efficient to use Deep Learning methods to train agents rather than manually implementing their behavior?

1.3 Delimitations

To limit the scope of the thesis, the agents will be trained on a set of different prototype simulations with down-scaled complexity. By disregarding higher levels of complexity not related to the task or the objective of the simulation, the agents face less noise in the training data and can therefore be expected to converge towards learning the actual objective of the simulation faster. Also, the purpose of this thesis is to evaluate how well the agents trained using Deep Learning can perform, and not to have a fully functioning product available for release.

The evaluation of the trained models will be performed by studying their ability to learn and by comparing their ability to maximize their received rewards to determine how rewarding their adapted behavior (or tactic) is. No in-depth analysis will be performed by visually studying the behavior of each agent, as it could greatly widen the scope of this thesis.

Also, the algorithms and techniques used will be based on earlier published research, and due to the technical limitations - no hyperparameter\(^1\) or network architecture optimization will be performed.

---
\(^1\)While parameters often refers to the weights of the network, hyperparameters refer to the parameters used in the algorithm while training the agent.
Chapter 2

Related Work

This chapter introduces previous research and work related to the subject of this thesis. It gives a short summary of methods where various Deep Learning methods are adapted to solve Reinforcement Learning problems, a domain referred to as Deep Reinforcement Learning. It also introduces how various methods of Machine Learning or Deep Learning have been used in the domain of military simulations.
2.1 Deep Learning in Games

For a long time, one of the long-standing challenges of Reinforcement Learning (RL) was to control the agent directly through high-dimensional sensory inputs, such as vision or speech. In most RL applications, the features from the data had to be hand crafted, heavily relying on the quality of the feature representation.

Recent advancements in Deep Learning (DL) has led to breakthroughs in areas like Computer Vision and Speech Recognition, thanks to its ability to extract high-level features from raw sensory data. However, most current DL applications have required large amounts of hand-labeled training data, where RL on the other hand must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed.

Google DeepMind, one of the current world leaders in artificial intelligence research and applications [19], has successfully managed to demonstrate results where the agent could overcome many of the challenges of learning successful control policies from raw image data as input in 2013 [20][21]. In what is referred to as Deep Q-Learning (DQL), DeepMind used a Deep Q-Network (DQN) in the Arcade Learning Environment [22] to train the agent on Atari 2600 games, resulting in state-of-the art in six out of seven games tested, and even surpassing human experts on three of them.

Since then, various alternative methods and improvements have been published built as an extension of DQN or with additions added to it, such as a doubled Q-network [23], dueling network architectures estimating the action values through estimations of action advantage and state value [24] and even adding a Recurrent Neural Network (RNN) to incorporate a sequential observability (or "memory") [25].

However, these methods relied on computationally expensive methods, often requiring specialized hardware and therefore limiting their accessibility.

The most recent advancements in Deep Reinforcement Learning (DRL) shifts the demand for specialized hardware through asynchronous methods, where multiple parallel agents operate in multiple environments on a single machine while training a global model [26], a framework referred to as Asynchronous Deep Reinforcement Learning (ADRL). Not only does ADRL reduce the hardware demands to run the algorithms, but also introduces extended capabilities on how the models are trained, empowering new algorithms that greatly surpasses any previous state-of-the art result in even shorter time. Due to the robustness of ADRL methods, the algorithms also succeed on various continuous motor control problems as well as on a previously unsolved task - navigating through random 3D mazes using a visual input.
2.2 Machine Learning Methods in Military Simulations

Because of its resource and time efficiency, it is common within the military to use simulations for training and educating purposes. The environments of these simulations are often rich and complex, and is populated with autonomous actors with specific roles [27]. Today, most autonomous actors are manually implemented with domain experts in a time and resource consuming process, often resulting error prone and predictable behavior [28].

The Swedish Defence Research Agency (FOI), in cooperation with several international agencies, is currently researching and developing Machine Learning tools and methods to simplify the development of synthetic actors for military simulations through Data-Driven Behavior Modeled Computer Generated Forces (CGFs) [29][30]. CGFs are autonomous, or semi autonomous, entities that typically represent military units, such as tanks, soldiers, fighter-jets etc. They have been used in military, simulation based, training and decisions for many decades.

A study by the Swedish Defence Research Agency compares the difference between manually implemented actors with actors trained through Machine Learning methods [31]. In the study, they used an Evolutionary Computing method known as Genetic Programming, where the program controlling the agent is considered an individual within a population of individual that is evolved and mutated through several generations. The study resulted in a genetically programmed agent that outperformed several professional programmers’ manual implementations by far.

Another study by the Dutch Ministry of Defence and Netherlands Aerospace Centre (NLR) studied the application of DL (or more specifically, DRL) methods of training an agent in Air Combat Behavior [32]. In their study, they trained an autonomous aircraft in air combat against another manually implemented opponent. Their result proved that the agent successfully learns to perform according to the objective and concluded that the application of DL and DRL should be further investigated for military simulation.

The Swedish Defence Research Agency has previously studied DL, but not in domain of training an agent to perform an objective [33].
Chapter 3

Theory

The following chapter describes some basic paradigms and learning algorithms of Machine Learning, and in particular Deep Learning. Initially Reinforcement Learning is explained, and then Artificial Neural Networks and variations of it such as Convolutional Neural Networks and Recurrent Neural Networks. Finally, this chapter describes Deep Learning, the benefit of using deeper architectures. Some state-of-the-art Deep Reinforcement Learning algorithms, that will be used in the latter experiments, are described and specified.
3.1 Reinforcement Learning

Reinforcement Learning (RL) [34] originates from the early days of cybernetics and work in areas such as Computer Science, Neuroscience, Psychology and Statistics. RL deals with the problem in where an agent must learn a specific behavior through trial-and-error interactions in order to solve a task in a dynamic environment. The RL problems are mainly solved using two different strategies. The first one, in which a space of behaviors is searched through to find one that would perform well in the environment, approaches the problem using for example Evolutionary Algorithms [35]. The other strategy is to approach the problem by using methods of estimating the utility of taking actions in states of the environment through statistical techniques and dynamic programming methods. In the objective of this report, the latter method is the most relevant and therefore will be the method explained further.

The environment in RL is formally described as a Markov Decision Process (MDP) [36]. The MDP contains a set of states $S$, a set of actions $A$, a reward function $R(s_t, a_t)$ and transitions $T$ between the states. At each time step of the process, the agent is in some state $s_t$. Given action $a_t$, it transitions into a successor state $s_{t+1}$ and receives a corresponding known scalar reward (or reinforcement signal) $R(s_t, s_{t+1})$. The state transition of a first order MDP can be defined according to the Markov Condition in equation (3.1), meaning that the state $s_{t+1}$ depends only on the previous state $s_t$ and thus is independent from all the states before $t$ [37].

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_1, ..., s_t) \quad (3.1)$$

The transitions between the states of an MDP can be represented as [38]:

- $T : S \times A \rightarrow S$
  *Deterministic*, where a new state is specified from the previous state and action.

- $T : S \times A \rightarrow \mathbb{P}(S)$
  *Stochastic*, for each state and action, a probability distribution over next state is specified as $P(s_{t+1}|s_t, a_t)$.

The core problem of an MDP, which RL is a method of solving, is to derive a *policy* $\pi$ for the agent. That policy is used to determine an estimate of the best possible action to perform in given state $s_t$ (defined as $\pi(s_t) = a_t$) in order to maximize the cumulative long-term reward [38].

The illustration in figure 3.1 visualizes the components of RL, where the agent (illustrated in figure 3.2) is connected to its environment via action and perception [37]. It contains the following components [39]:

- A *reward function*, which defines the goal and the behavior of the agent by reinforcing the value specific states.
• A policy \((\pi)\), which is the decision making function of the agent that specifies which action to execute in each state it encounters such as to take high value actions in order to maximize the rewards over time.

• A value function, specifying the value of how good a state is in terms of expected future rewards form a state given a policy.

![Figure 3.1: The Agent-Environment Interaction model.](image)

Through these, the agent can derive its policy without knowing about the reward function or possible transitions by interacting with the environment and using methods of assigning values to states.

### 3.1.1 Value Iteration

A method of deriving a policy from a standard MDP is through value iteration, where an optimal policy is found by finding an optimal value function \([40]\). In value iteration, all states have arbitrarily initialized values \(V(s_i)\). That value is then recursively estimated to \(V(s_t)\) based on the value of the successor \(V(s_{t+1})\) for action \(a\) and reward \(r\) \([34]\). This way, a backup of each estimated state value is kept in a table, and the Bellman Equation \([41]\) is then used to update it. The update rule uses a discounted \((\gamma \in (0,1))\) sum of the future rewards, the reward \(r(s_t)\) and value of the successor state \(V(s_{t+1})\) to update \(V(s_t)\) \([42]\).
\[ V(s_t) \leftarrow r(s_t) + \gamma \max_a \sum_{s_t} P(s_{t+1}|s_t, a) V(s_{t+1}) \] (3.2)

Depending on if the reward function is known or not - value iteration is performed differently. In the case where the reward function is known, value iteration is used to calculate the values of the states in order to find the optimal policy. But when the reward function is not known, and the estimate of it must be discovered by exploration of the states.

### 3.1.2 Actor-Critic

Actor-Critic is an extension of value iteration, where there is a separate memory structure for the policy and value function independent of each other [43]. The policy \( \pi \), or the Actor, is used to select actions and the value function, or the Critic, criticizes the actions made by the agent [37]. Another way of describing it is that the Actor determines how to make decisions, thus representing a short-sighted strategy of only looking one state ahead, while the Critic is a more long-term, strategic, evaluation of the agent on how valuable the current state is (i.e. if it currently is "winning or losing") [44].

For an agent selecting the action \( a \) in state \( s_t \), the error of the value function is calculated using equation (3.3). If the error is positive, the tendency to select action \( a \) in state \( s_t \) is strengthened, if it is negative, it should be weakened.

\[ E = r_{s_t} + \gamma V(s_{t+1}) - V(s_t) \] (3.3)

Using \( p(s_t, a_t) \) as modifiable policy parameters of the agent, the update equation for increased or decreased tendency to select an action can be written as in equation (3.4), where \( \beta \) is a positive step-size parameter [37].

\[ p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta E \] (3.4)

Actor-Critic algorithms can thus benefit from both the Actor and Critic being evaluated at the same time, and as the Critic also updates the Actor, it results in gradients with a lower variance and is expected to speed up the learning process and converge faster [45][46].
3.1.3 Q-Learning

One of the most popular and most effective model-free algorithms in RL is Q-Learning. It is based on a combination of value iteration and Adaptive Heuristic Critic [47], and is categorized as a one-step Actor-Critic method. In Q-Learning, the idea is to define a Q-function, setting a value (or Q-value) to each state-action pair. Like in value iteration (see section 3.1.1), all Q-values are initialized arbitrarily and are then iteratively updated as the agent transitions between states.

$Q(s, a)$ defines the expected discounted reward (or reinforcement) when an action $a$ is taken in the state $s$, acting as a critic to the agent and its actions. The Q-value basically says how good a certain action is in a given state. The agent uses the simple policy of performing the action with the highest Q-value (see equation (3.5)) in order to maximize its future reward. This is an off-policy method as it estimates a return for state-action pair assuming that a greedy policy, as in taking the action with the highest Q-value, is followed[37].

The Q-value of each state is guaranteed to converge to its actual value, $Q^*$, given an infinite number of action executions on each state.

$$\pi(s) = \max_a Q(s, a) \quad (3.5)$$

The estimated Q-value, based on the Bellman Equation [37] (equation (3.6)), is iteratively updated using equation (3.7) as the agent explores the environment with a discount value $\gamma \in (0, 1]$.

$$Q(s_t, a_t) = r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (3.6)$$
Since Q-Learning updates the Q-values based on which actions it takes and which state that leads it to, you want the agent to explore as much as possible of the environment. But at the same time, you want the agent to explore the nearby areas with a high Q-value - which leads to a balance between exploration and exploitation [34]. An exploration rate, $\epsilon \in [0, 1]$, determines how often agent sticks to its greedy policy (equation (3.5)) and taking the action that is estimated to return the highest cumulative future rewards, or takes a random action regardless of its policy.

This way of exploring the environment allows the agent to potentially find new rewarding states, or different transition routes of reaching the rewarding state - that might be shorter than the previously found path [37]. See the full algorithm and details of Q-Learning in algorithm 1.

Algorithm 1 Q-Learning algorithm

1: Initialize $Q(s_t, a_t)$ arbitrarily
2: for each episode do
3: Initialize $s_t$
4: for each step of episode, until $s$ is terminal do
5: Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
6: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta(r_t + \gamma \max_{a_{t+1}}Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
7: end for
8: end for

3.2 Artificial Neural Networks

The human brain has an exceptional ability to learn, memorize and still generalize like a complex, nonlinear and parallel computer using networks of biological neurons. Neurons of the network are connected via synapses to other neurons, and if the input of one neuron surpasses a certain threshold, it transmits an electrical or chemical signal to the other connected neurons [48]. Inspired by this system, researchers have attempted to mimic it in an artificial model, with artificial neurons interconnected via weights. The resulting model is referred to as an Artificial Neural Network (ANN), also known as a Feed-Forward Network. ANNs are still used today in the leading research of many areas, especially Deep Learning (see section 3.5). Various methods of training ANNs in larger scale are still being developed and refined [49].

Modeled to operate like the biological neuron, each Artificial Neuron (AN) (neuron, node, or unit) collects one or more signals from an environment or another AN. It then computes a net input signal as a function of the
respective weights, which then serves as an input to its activation function, calculating the output of the AN [39] (see example model in figure 3.4).

![Figure 3.4: An Artificial Neuron with 3 inputs.](image)

Given the sigmoidal activation function in equation (3.8) (where λ defines the horizontal asymptotes), the output of an AN $y$ is calculated through the sum of the inputs $x_i$ and their respective weights $w_i$ through the activation function $S$ (see equation (3.9)).

$$f(S) = \frac{1}{1 + e^{-\lambda}} \quad (3.8)$$

$$y = f(S) \quad \text{where} \quad S = \sum_{i=0}^{n} w_i x_i \quad (3.9)$$

An ANN consists of a network of ANs, normally architecturally organized into layers of units where each layer output is the next layers input [50]. The first layer is the input layer, the layers in between (one or more) are called hidden layers and the final layer is the output layer (see example model in figure 3.5). The ANs within the network are fully connected to the adjacent layer with weighted connections - making the ANN a family of neurons parameterized by their weights [51].

ANNs with at least one hidden layers are considered universal approximators. This means that, given any continuous function $f(x)$ and accuracy $\epsilon > 0$, there is an ANN $g(x)$ that can approximate $f(x)$ it (expressed mathematically in equation (3.10)) [51][52].

$$\forall x, |f(x) - g(x)| < \epsilon \quad (3.10)$$
Due to the ANNs great efficiency and ability to solve complex problems, the classes of applications they are used in today include classification, pattern completion, control and optimization [39], covering paradigms such as supervised learning, unsupervised learning and reinforcement learning.

### 3.2.1 Training an Artificial Neural Network

Since ANNs are parameterized by their weights, they are trained by adjusting these weights such that the error $E$ is minimized. A common method is to backwards propagate the error through the network (also known as backprop) in conjunction with a method of optimization such as gradient descent.

Backprop uses the error $E$ between the expected output $y$ and the actual output $\hat{y}$, and uses the chain rule to iteratively compute the error gradient for each layer’s weights $w_i$ using a learning rate $\eta$ (see example for single neuron in equation (3.11)) [53].

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial S} \frac{\partial S}{\partial w_i} = -(\hat{y} - y)f'(S)x_i$$

That gradient is then used to update the weights of that layer using gradient
descent methods. Common method is Stochastic Gradient Descent (SGD) seen in equation (3.12) [54]:

$$
\Delta w_i = -\eta(\hat{y} - y)f'(S)x_i
$$  
(3.12)

Another common method is to compute the gradient over $n$ samples referred to as Mini-batch Gradient Descent (MGD).

These methods alone have some difficulties attached to them, such as finding a good learning rate $\eta$ or getting trapped in a local minimum of the error gradient. However, there are certain algorithms to performing the gradient descent that have methods of achieving better convergence. Examples of these algorithms are momentum based Gradient Descent [55], RMSProp [56] and Adam [57].

### 3.2.2 Combining Artificial Neural Networks and Reinforcement Learning

Some MDP problems with a small-enough state-space can be solved with a simple look-up table containing e.g. state values or state-action pairs for each state. However, as the state space and complexity of the problem increases, such as in a dynamic environment, this method becomes unfeasible and impractical [20]. A better approach would be to generalize and pattern match between the states, so that algorithm learns to find similarities between states and takes action thereafter [37].

Here, where ANNs can be used as a function approximator in combination with RL to improve performance of the agent [58] by being trained on estimating the values or state-action pairs for each state instead of storing it in a table. The following sections will further discuss similar methods of combining RL and ANNs, and how they ultimately are used in the experiments in this thesis.

### 3.3 Convolutional Neural Networks

A Convolutional Neural Network (CNN) has many similarities with an ANN, both functionally and architecturally [59]. They both consist of neurons, have weights and biases it can learn. Inspired by mimicking the organization of an animal visual cortex, where the individual neurons are arranged so that they respond to specific overlapping regions of a visual field [60], the CNN has a spatial structure where specific regions of the layer are connected to the units in the next layer.

The neurons of a CNN are therefore arranged as a volume, with a width, height and depth. Its layers are commonly divided into three categories: convolutional layer, pooling layer and a fully connected layer, through which
the volume of activation is transformed from one layer to another using a differentiable function.

Figure 3.6: An illustrated example of a Convolutional Neural Network and how it operates with a volume of neurons in three dimensions. The leftmost layer represents the input with a width, height and depth, followed by two layers of three dimensional convolutional operations and ultimately a fully connected layer that connects to an output.

3.3.1 Convolutional Layer

The convolutional layer consists of a set of small set of learnable filters (or kernels) that will learn to activate when they see some visual feature, such as an edge or a specific color [60]. This is performed by spatially convolving (or sliding) each filter across the width and height of the input volume (with a fixed stride length). At each slide, the dot product between the input and the entries of the filter is computed that ultimately produces an output commonly referred to as a feature map, or activation map [59]. An illustrated example in figure 3.7 shows two steps of how a 7x7 input is being convolved with a 3x3 filter and stride length 1, resulting in a 5x5 feature map (or output). In figure 3.8, the illustrated example shows how the output differs if stride length is changed to 2, resulting a 3x3 feature map. Not that the stride also has an effect the length of the vertical slide.

Figure 3.7: An illustrated example of a horizontal slide in convolution on a 7x7 input using a 3x3 filter map and stride length 1.
Figure 3.8: An illustrated example of a horizontal slide in convolution on a 7x7 input using a 3x3 filter map and stride length 2.

An actual example of different filters being applied the same input image can be seen in figure 3.9 [61], showing the various resulting outputs depending on the filter used.

\[
\begin{bmatrix}
1 & 0 & -1 \\
0 & 0 & 0 \\
-1 & 0 & 1
\end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}, \begin{bmatrix} -1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1
\end{bmatrix}
\]

Figure 3.9: Examples of an input image (top) run through various filters (middle) for edge detection, and their respective output (bottom).

3.3.2 Pooling Layer

Sometimes, in order to reduce the number of parameters and computation in the network, the output of the convolutional layer can be down-sampled in a pooling layer. By commonly using a mean or max value within a sub-region of the Convolutional Layer output, the pooling layer reduces the spatial size of the representation and potentially improves the result by minimizing the chance of over-fitting. The operation of the pooling is visually demonstrated in figure 3.10 [59] and figure 3.11.
Figure 3.10: An illustration showing how pooling down samples the width and height of a volume while keeping spatial information and the input volume.

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
2 \\
5 \\
8 \\
\end{array}
\rightarrow
\begin{array}{c}
4 \\
3 \\
5 \\
9 \\
2 \\
\end{array}
\rightarrow
9
\end{array}
\end{array}
\]

Mean-pooling example  Max-pooling example

Figure 3.11: Examples of two different pooling techniques.

Despite of the positive properties of the pooling layer, it is not an essential part of the network architecture. It can, for example, be replaced with more convolutional layers without loss in accuracy according to several image recognition benchmarks [62].

3.3.3 Fully Connected Layer

Much like in regular ANNs the fully connected layer has full connections to all the units in the adjacent layers, and hence are computed with matrix multiplication and a bias offset. Since this section of the network architecture is fully connected, it drops the spatial structure of the previous layer and can therefore be visualized in one dimension. As an effect of the deconstruction (or loss) of the spatial structure, there can be no additional convolution layers after this section, and therefore the fully connected layer commonly outputs the output of the whole network architecture [63].

3.4 Recurrent Neural Networks

Even though an ANN or a CNN imitate some capabilities of the human brain, they lack the major feature of the biological brain that gives us a higher level of context awareness - some form of sequential memory. For example, a CNN could potentially give each frame in a movie a given tag or label but would not be able to determine what the current events of the specific scene (or series of frames) are. In the same way, a single word does not make sense to us human, because we would need the word in a context,
such as a sentence. A Recurrent Neural Network (RNN) addresses this issue by allowing information to persist within the network, acting like a limited amount of memory where the output is dependent on a chain of inputs [64].

The unfolded structure of a RNN (see figure 3.12) reveals several copies of the same network, where the output of one unit is passed on to the input of the next unit. Studying a single RNN unit with an input $x$ and output $h$, the illustration in figure 3.13 shows how an input from a previous unit ($h_{t-1}$) and new input ($x_t$) is passed through a tanh-operation, ultimately producing the output $h_t$ that also is passed on to the next unit.

This looped structure allows RNNs to perform well in tasks such as Natural Language Processing [65], Speech Recognition [66] and Machine Translation [67].

Figure 3.12: Recurrent Neural Network model, folded (left) and unfolded (right) with sequence of length $n$. 
3.4.1 Long Short-Term Memory

Despite RNNs ability to learn a sequence of inputs, it can suffer from difficulty if the dependencies of the input are far apart. For example, in Language Modeling where sentence completion predicts the next word depending on the previous words, the influence of the early words might decay as they pass through the unfolded chain. For a human, it might be obvious that the sentence “I grew up in France and I speak fluent...” should be followed by “... French.”, but in practice, the RNN might have trouble with coming up with that prediction because the impact of the best clue (“... France ...”) might have decayed over the sequence [64].

A different architecture of RNNs that is more resilient to dependencies that appear far apart is the Long Short-Term Memory (LSTM) [68]. By determining the significance of the input, the LSTM can pass on valuable information in between its units, called cells (or memory cells), so that it does not decay along the sequence of cells [69]. An illustration in figure 3.14 shows an example of the structure and operations within an LSTM cell.
Figure 3.14: The structure and operations of a Long Short-Term Memory cell.

From left to right in figure 3.14, the first sigmoid ($\sigma$) layer of the LSTM cell calculates how much of the older output ($h_{t-1}$) to forget using equation (3.13), also known as the “forget gate layer”. For example, $f_t = 1$ means that everything should be remembered, while $f_t = 0$ means that everything should be forgotten.

$$f_t = \sigma(W_f \ast [h_{t-1}, x_t] + b_f) \quad (3.13)$$

It then determines what to store in the cell state ($C_t$), first using another sigmoid layer, but also a $\tanh$ layer using equation (3.14), and then finally updates the old cell state ($C_{t-1}$) into the new cell state ($C_t$) using equation (3.15).

$$i_t = \sigma(W_i \ast [h_{t-1}, x_t] + b_i) \quad (3.14)$$
$$c_t = \tanh(W_C \ast [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t \ast C_{t-1} + i_t \ast c_t \quad (3.15)$$

Finally, the output is calculated based on the filtered cell state ($C_t$) and the input run through the last sigmoid layer using equation (3.16).

$$o_t = \sigma(W_o \ast [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \ast \tanh(C_t) \quad (3.16)$$

LSTMs unique characteristics has allowed them to solve many previously unsolvable tasks [49], and is today state-of-the-art in a wide range of research areas and applications, such as image generation [70], text-to-speech synthesis [71] and social signal classification [72], but also in areas like medicine [73] artificial creativity and art [74].
3.5 Deep Learning

A major source of difficulty in Machine Learning has been to manually extract general high-level, abstract features from raw data, such as the similarities of the same face from two different angles or the color of a car under different lightning conditions. The theory of Deep Learning (DL) can be dated back to the 60’s [11], but due to the lack of training data, robust algorithms handling problems such as diminishing gradients [75] and computational limitations it has not been until recently that the advancements in the field have started accelerating.

DL represents a sub-field of Machine Learning that enables solving more complex problems by using a deeper network architecture [76]. The added depth can enable the network to learn higher-level representation through lower-level features from the input, and the lower-level features can be used in several representations of higher-level feature abstractions [77] (see figure 3.15 [78] for an example). This allows the network to represent functions with increasing complexity [76]. Commonly, DL networks use a combination of different neural networks, such as ANNs, CNNs, RNNs or LSTMs.

The performance of using DL algorithms in areas like Computer Vision, Speech Recognition, Natural Language Processing and Bioinformatics has led to state-of-the-art results [79][80][81].

Figure 3.15: Hierarchical features of facial recognition, showing how the features learned to go from edges, facial features and ultimately to full faces with the depth of the network (left to right).

3.5.1 Deep Reinforcement Learning Methods

Problems where Deep Neural Network (DNN) architectures are applied to a RL are commonly referred to as Deep Reinforcement Learning (DRL). Using the approach of a Neural Network in a RL environment can have advantages (as described in section 3.2.2), and using a DNN has the potential to further increase those advantages. The reason for the potential increase in performance is due to the increased ability to find higher-level representation features through the added depth, e.g. through a CNN [21].
3.5.1.1 Deep Q-Learning

One of the techniques of DRL is a method called Deep Q-Learning (DQL) [20]. It estimates the Q-values and weights $Q(s, a; \theta)$ using 4 stacked image frames (or screenshots) with raw pixels as an input and outputs a set of Q-values corresponding to the number of actions available in the environment (see figure 3.16 for network architecture).

![Deep Q-Network Architecture](image)

Figure 3.16: A Deep Q-Network Architecture example with outputs corresponding four different actions.

By using a technique called Experience Replay, the agent’s experiences are stored at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set and pooled over many episodes into a replay memory $D$. Over the loop of iterating the algorithm, the Q-values are updated using mini batches of randomly sampled experiences from this data set. This has an advantage compared to standard Q-Learning since every experience of the agent is used in many weight updates, causing much less weight oscillations and better smoothing as well as better data efficiency [82].

Using the same network for taking actions and training had previously proven to be unstable. At every step of the training, the estimated Q-values of the network could shift, and if constantly shifting set of values are used to update the parameters of the network, the estimations would often lead into feedback loops and spiral out of control [83]. A method of countering this is to use two networks, a target network (which is used to estimate the Q-values) and an online network that is trained. The weights of the target network ($\theta^-$) are kept fixed for a number for steps $n$ while the online network $Q(s, a; \theta_i)$ is updated. Every $n$th step, the target network would then be replaced with the updated online network [24]. This has a stabilizing effect since using one single network to both train and

In Deep Q-Network (DQN), the optimal state-action value function $Q^*$ is estimated using the function based on Bellman Equation for the optimal
action-value function (equation (3.17)).

\[
Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1}} [r + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s_t, a_t] \tag{3.17}
\]

By representing the network with weights \( \theta \), as in equation (3.18), the loss is computed using the loss function in equation (3.19) using past Experiences from the Experience Replay (D) [82].

\[
Q(s_t, a_t; \theta) \approx Q^*(s_t, a_t) \tag{3.18}
\]

\[
y^{DQN} = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^-)
\]

\[
L(\theta_i) = \mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim D} [(y^{DQN} - Q(s_t, a_t; \theta))^2] \tag{3.19}
\]

See the full algorithm and details of DQN explained in pseudocode in algorithm 2.

**Algorithm 2** Deep Q-Learning with Experience Replay

1. Initialize Experience Replay memory \( D \) to capacity \( N \)
2. Initialize network with random weight parameters \( \theta \) and target network with a copy of them, \( \theta^- \)
3. for episode = 1, \( M \) do
4. Initialize sequence \( s_1 = \{x_1, x_1, x_1, x_1\} \)
5. for \( t = 1, T \) do
6. With probability \( \epsilon \) select a random action \( a_t \)
7. otherwise select \( a_t = \text{argmax}_a Q^*(s_t, a_t, \theta) \)
8. Execute action \( a_t \) and observe reward \( r_t \) and input \( x_{t+1} \)
9. Set \( s_{t+1} = s_t, a_t, x_{t+1} \)
10. Store transition \( (s_t, a_t, r_t, s_{t+1}) \) in \( D \)
11. Sample random mini-batch of transitions \( (s_j, a_j, r_j, s_{j+1}) \) from \( D \)
12. Set \( y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a_j} Q(s_j, a_j; \theta^-) & \text{for non-terminal } s_{j+1} \end{cases} \)
13. \( \alpha = (y_j - Q(s_j, a_j, \theta))^2 \)
14. Perform gradient descent step on \( \alpha \) according to equation (3.19)
15. end for
16. end for

### 3.5.1.2 Dueling Deep Q-Learning

An approach that implements the previously mentioned Actor-Critic method (section 3.1.2), but in a similar approach to the DQL is the Dueling Deep Q-Learning (DDQL) [24]. In DDQL, the advantage function \( A(s_t, a_t) \) is introduced [84]. Since \( Q(s_t, a_t) \) represents the value of action \( a \) in state \( s \), \( V(s_t) \) represents the value of that state independent of the action. With the definition \( V(s_t) = \max Q(s_{t+1}, a_{t+1}) \), the advantage \( A(s_t, s_t) \) is then a
relative measure of utility of actions in \( s \) through equation (3.20) \[85\]. In some states, it is of great importance to know which action to take, but in others - it has no repercussion on of what happens. Therefore, the dual network architecture can potentially improve the performance of the agent since it isn’t fully dependent on estimating single state-action pairs but also estimates the value of the state.

\[
A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{3.20}
\]

A Dueling Deep Q-Networks (DDQN) architecture contains a CNN at its upper levels, but is then separated into two streams of fully connected layers, providing separate estimates for advantage and value functions (see figure 3.17). The estimates of these two networks is ultimately combined to produce a set of Q-values using equation (3.21).

\[
Q(s_t, a_t) = V(s_t) + A(s_t, a_t) \tag{3.21}
\]

The network is trained with parameters for the CNN \((\theta)\) and the parameters of the two streams \((\alpha \& \beta)\). In order to improve stability, the advantage function estimator is forced to have zero advantage at the chosen action (see equation (3.22)).

\[
Q(s_t, a_t; \theta, \alpha, \beta) = \left[ V(s_t; \theta, \beta) + \left( A(s_t, a_t; \theta, \alpha) - \max_{a_{t+1}} A(s_t, a_{t+1}; \theta, \alpha) \right) \right] \tag{3.22}
\]

The DDQN also uses Experience Replay with the replay memory in the same manner as the DQN described in section 3.5.1.1 in order to improve training stabilization and data efficiency of training the agent.

Figure 3.17: A Dueling Deep Q-Network Architecture example with outputs corresponding four different actions.
This architecture of DDQN should lead to better policy evaluation in the presence of many similar-valued actions since it can learn which states are (and are not valuable), without having to learn of each action for each state. This has been proven to be useful in states where its actions do not affect the environment in any relevant way [24].

3.5.2 Asynchronous Deep Reinforcement Learning Methods

The combination of online RL algorithms and DNNs had been proven to be unstable, which is why methods were introduced with the purpose of stabilizing the learning (e.g. Experience Replay, presented in section 3.5.1). But those methods also introduce drawbacks, such as an increased amount of memory usage and an off-policy learning method that can update from data generated by an older policy, relying heavily on specialized hardware or distributed architectures. Also, the models take very long to train using these algorithms with the added stabilization methods.

Asynchronous Deep Reinforcement Learning (ADRL) introduces a method of countering many of those drawbacks by executing multiple, asynchronous, agents in parallel on multiple instances of the same environment. The agents share a global model, but act on and train a copy of it for a fixed number of steps before updating the global model and receiving a new copy of it (every $I_{update}$ steps) [26]. The asynchronous agents are likely to be exploring different parts of the environment of the same time, which potentially could converge faster and have a stabilizing effect on the training process. This also enables the usage of online learning algorithms such as Q-Learning and Actor-Critic methods instead of using one online network and a regularly updated target network (like in DQL and DDQL methods).

Asynchronous methods have a roughly linear effect on the reduction of training time with respect to number of parallel agents. Also, they use less computational resources and can run on standard multi-core CPU-equipped hardware [26].

3.5.2.1 Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) is the multi-threaded asynchronous variant of an Actor-Critic method that maintains a policy and a value estimate of the through a set of asynchronous agents. Similarly to DQL, it uses a CNN in the upper levels leading to fully connected layer. The output of the fully connected layer is used for both estimating the policy $\pi(a_t|s_t; \theta)$ and the value function $V(s_t; \theta)$.

The value and the policy function are then updated with an accumulated error gradient after every $t_{max}$ steps or when the agent reaches a terminal state. The update is performed using equation (3.23), where $A(s_t, a_t, \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) -$
$V(s_t; \theta_v)$ and where $k$ increases with each state and is upper-bounded by $t_{\text{max}}$. The gradient of the full objective function, including the regularization term of the entropy with respect to the policy parameters can be seen in equation (3.24), where $R_t$ is an estimate of $Q^\pi(a_t, s_t)$, $H$ is the entropy and $\beta$ controls the strength of the entropy regularization term. See the full algorithm of a single thread A3C agent explained in pseudocode in algorithm 3 [26].

$$\nabla_{\theta'} \log \pi(a_t|s_t; \theta') \ A(s_t, a_t, \theta, \theta_v)$$  
(3.23)  

$$\nabla_{\theta'} \log \pi(a_t|s_t; \theta')(R_t - V(s_t; \theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t; \theta'))$$  
(3.24)

Recent studies by Minh et al. [26] shows that the A3C method not only surpasses most state-of-the-art DRL methods on the Atari domain, but often does it with a large margin and in a potentially shorter time as well. An illustration showing an example of the network architecture can be seen in figure 3.18.

![Figure 3.18: Example of an A3C Network Architecture with a Feed-Forward Neural Network.](image)

### 3.5.2.1.1 LSTM Network Architecture

The basic approach of the A3C network architecture uses a regular feed forward network (in this report referred to as an ANN) according to the illustration in figure 3.18. However, in an attempt to extend its capabilities and performance, research form Minh et al. [26] also shows that adding an additional 256 LSTM Cells (see section 3.4.1) after the fully connected units greatly improved the performance of the algorithm.
Algorithm 3: Asynchronous Advantage Actor-Critic - single thread algorithm

1: Assume global shared parameter vectors $\theta$, $\theta_v$, and global shared counter $T = 0$
2: Assume thread-specific parameters $\theta'$ and $\theta'_v$
3: Initialize thread step counter $t \leftarrow 1$
4: repeat
5:   Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
6:   Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
7:   $t_{start} = t$
8:   Get state $s_t$
9: repeat
10:   Perform $a_t$ according to policy $\pi(a_t|s_t; \theta')$
11:   Receive reward $r_t$ and new state $s_{t+1}$
12:   $t \leftarrow t + 1$
13:   $T \leftarrow t + 1$
14: until terminal $s_t$ or $t - t_{start} = t_{max}$
15: $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$ // Bootstrap from last state
16: for $i \in \{t - 1, ..., t_{start}\}$ do
17:   $R \leftarrow r_i + \gamma R$
18: Accumulate gradients w.r.t.:
19:   $\theta' : d\theta' \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_t|s_t; \theta')(R - V(s_t; \theta'_v))$
20:   $\theta'_v : d\theta_v \leftarrow d\theta_v + \partial R - V(s_t; \theta'_v)^2 / \partial \theta'_v$
21: end for
22: Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$
23: until $T > T_{max}$
Chapter 4

Method

The following chapter specifies a more detailed view of the Deep Reinforcement Learning algorithms that were used to train the models and explains why they were selected. It also includes a detailed view of the parameters and settings used during the training for both the algorithm and the simulation environment, and how the trained models will be evaluated afterwards. The chapter also includes the details of each specific experiment and explains both their respective objectives and the purpose of their setup.
4.1 Experiment Methodology

4.1.1 Experiment Simulations

To perform the experiments of this thesis, four different simulations (one for each experiment) were produced; simulating events or objectives that actor of Computer Generated Forces (CGFs) could potentially face in a military setting.

Each simulation has a different objective, level of complexity and different tactics that could be adapted in order to perform the objective of it. The diversity of them will contribute to get a broader view in order to draw the conclusions for the problem statement (section 1.2.1) of this thesis. The specifics of each experiment (and simulation that was used in it) can be found in section 4.2.

4.1.2 Algorithms

A total of four different Deep Learning (DL) algorithms were selected to train the agents with in the experiments. This limited subset is far from the whole spectrum of different methods, even within Deep Reinforcement Learning (DRL), but are selected for their proven good performance in experiments previously conducted by from Mnih et al. [20][26] and Wang et al. [24]. They represent the two different categories within DRL algorithms: single-agent DRL (section 3.5.1.1) and multi-agent DRL (section 3.5.2), hereby referred to and categorized as Regular Agent (RA) and Asynchronous Agent (AA) methods (or models).

The selected algorithms are:

- **Deep Q-Network (DQN)** with Experience Replay, because of it’s simple approach of Q-Learning and network structure in combination with Experience Replay.

- **Dueling Deep Q-Networks (DDQN)** with Experience Replay, because it utilizes a similar approach to Deep Q-Learning (DQL) with Experience Replay, but with both a value network and an advantage network, giving it the ability to act upon both advantage and value.

- **A3C with a Feed Forward Network (A3C-FF)**, because it represents an asynchronous version of an Actor-Critic method. This, multi-agent approach should have a better chance of converging and result in better performance due to the large state space of the simulations the agent is trained in.

- **A3C with an LSTM Network (A3C-LSTM)**, mainly for the same reason as using the A3C-FF, but also to study the difference in performance when the network utilizes LSTM cells in addition to the regular units with an activation function.
4.1.3 Algorithm and Simulation Settings

The algorithms used in the experiments have almost infinite possible hyper-parameter combinations to use in training the agent. However, due to the limitations of the extent of this thesis, a fixed setting will be used across all the experiments and simulations. This means that the results are not representing the absolute ground truth of an optimized agent performance but will be used as a guideline to evaluating the performance. These hyperparameters are based on previous research by Mnih et al. [20][26] and Wang et al. [24], which the exception of changes in the global max time step \(T_{\text{max}}\), which has been reduced by half for both RA and AA in order to limit training-time for the model.

The hyperparameters of the RA methods can be viewed in table 4.1.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max timestep (T_{\text{max}})</td>
<td>5 000 000 steps</td>
</tr>
<tr>
<td>Exploration anneal (\epsilon_{\text{anneal}})</td>
<td>1 000 000 steps</td>
</tr>
<tr>
<td>Exploration (\epsilon)</td>
<td>1 (annealed to 0.1 over (\epsilon_{\text{anneal}}))</td>
</tr>
<tr>
<td>Discount factor (\gamma)</td>
<td>0.99</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.00025</td>
</tr>
<tr>
<td>Experience Replay size</td>
<td>1 000 000</td>
</tr>
</tbody>
</table>

Table 4.1: Hyperparameters for the single-agent methods.

The hyperparameters of the AA can be viewed in table 4.2.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global max timestep (T_{\text{max}})</td>
<td>50 000 000 steps</td>
</tr>
<tr>
<td>Exploration (\epsilon)</td>
<td>0.1</td>
</tr>
<tr>
<td>Parallel agents</td>
<td>16</td>
</tr>
<tr>
<td>Discount factor (\gamma)</td>
<td>0.99</td>
</tr>
<tr>
<td>Learning rate (Actor error)</td>
<td>0.0007 (annealed to 0 over (T_{\text{max}}))</td>
</tr>
<tr>
<td>Learning rate (Critic error)</td>
<td>0.00035 (annealed to 0 over (T_{\text{max}}))</td>
</tr>
<tr>
<td>Local network update period (t_{\text{max}})</td>
<td>5 steps or terminal state</td>
</tr>
<tr>
<td>Global network update period (I_{\text{update}})</td>
<td>5 steps</td>
</tr>
<tr>
<td>Entropy Regularization constant (\beta)</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 4.2: Hyperparameters for the asynchronous multi-agent methods.

The simulation in which the agent was trained had a set of general settings across the experiments, with some settings changed in the latter experiments to introduce a larger state-space and increase complexity of the experiments. Unless otherwise is stated, it can be assumed that the simulations had the settings according to table 4.3.
Also, a set terminal constraints have been added (that vary across the simulations) that terminate the current episode of the agent. Their purpose is to shorten the training time, helping the agent to converge faster and to limit the length of each episode.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment size</td>
<td>80x80</td>
</tr>
<tr>
<td>Max steps / episode</td>
<td>2000</td>
</tr>
<tr>
<td>Max non-rewarded steps (terminal state)</td>
<td>300</td>
</tr>
<tr>
<td>Agent step length</td>
<td>3 coordinate units</td>
</tr>
<tr>
<td>Agent action size</td>
<td>4 (Right, Left, Up, Down)</td>
</tr>
</tbody>
</table>

Table 4.3: General settings.

4.1.3.1 Network Architectures

The network architectures of this thesis have been based on set ups to of experiments by Mnih et al. [20][26] and Wang et al. [24]. Their details can be viewed in table 4.4. Note that for some experiments, where the action size was increased from four to five, the network output was also increased changed to five. Other than that, the network architectures were fixed for all across experiments.

An illustration of the DQN network architecture can be viewed in figure 3.16, the DDQN network architecture in figure 3.17, the A3C-FF in figure 3.18 and the A3C-LSTM in figure 3.19.
<table>
<thead>
<tr>
<th>Network</th>
<th>Input</th>
<th>Hidden layers</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>4 x 80x80</td>
<td>16 x 19x19 Feature map</td>
<td>Action output 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 x 8x8 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 Fully connected units</td>
<td></td>
</tr>
<tr>
<td>DDQN</td>
<td>4 x 80x80</td>
<td>Advantage network</td>
<td>Action output 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 x 19x19 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 x 8x8 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 Fully connected units</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value network</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 x 19x19 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 x 8x8 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 Fully connected units</td>
<td></td>
</tr>
<tr>
<td>A3C-FF</td>
<td>4 x 80x80</td>
<td>16 x 19x19 Feature map</td>
<td>Action output 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 x 8x8 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 Fully connected units</td>
<td></td>
</tr>
<tr>
<td>A3C-LSTM</td>
<td>4 x 80x80</td>
<td>16 x 19x19 Feature map</td>
<td>Action output 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 x 8x8 Feature map</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 Fully connected units</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 LSTM Cells</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value output</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.4: Network Architectures.

4.2 Experiments

4.2.1 Experiment 1: Collect Items

In order to have a baseline to compare the performance of the experiments overall, an experiment with a very simple objective was set up. The objective of this experiment was to maximize the number of collected items in a limited number of steps. At any given time step, the environment would contain five randomly placed items and whenever the agent collected any of them, a new would appear at a new random location within the environment. In order to pick up an item, the agent had to reach a distance of below 10 coordinate units from it.

The purpose of this experiment is to see if the agent can learn a simple task, but also to examine if the algorithms used to train it may result in different performance, revealing the agents’ abilities to prioritize how to collect of items. For example, if the agent can learn that it is beneficial to stay around the center of the environment to quickly be able to reach rewards around it.
See equation (4.1) for reward function of this experiment, and figure 4.1 for an illustrated sequence of the simulation. The environment in which the agent operated in had no obstacles or other parameters that would impact its performance on following the target.

\[
    r = \begin{cases} 
    1 & \text{distance}_{item,\text{agent}} < 10 \\
    -1 & \text{moving outside environment boundaries} \\
    0 & \text{otherwise} 
    \end{cases} \quad (4.1)
\]

Figure 4.1: An illustrated sequence of the first simulation where agent is rewarded by collecting items.

### 4.2.1.1 Terminal Constraints

This simulation had three different ways reaching a terminal state in an episode.

1. The agent made 300 consecutive steps without receiving any reward.
2. The agent attempted to move outside environment grid boundaries.
3. The agent reached a total of 2000 steps.

### 4.2.2 Experiment 2: Collect Items (With Obstacles)

This experiment aimed to increase the complexity and to see how the agent would learn the behavior of prioritizing in an area with obstacles. Just like in the experiment described in section 4.2.1, the environment would always contain a total of five randomly placed items, and whenever the agent reached any of them a new one would be generated at a new random location. In comparison, the required distance to pick up an item was half as big (previously 10 coordinate units, now 5), which would force the agent to move longer distances in order to pick up the items.

With respect to the placed obstacles, the reward function was also edited and can be viewed in equation (4.2). An illustrated sequence of the simulation can be viewed in figure 4.2.
\[ r = \begin{cases} 
1 & \text{distance}_{\text{item}, \text{agent}} < 5 \\
-1 & \text{moving outside environment boundaries} \\
-1 & \text{moving against an obstacle} \\
0 & \text{otherwise}
\end{cases} \quad (4.2) \]

Figure 4.2: An illustrated sequence of the second simulation where agent is rewarded by collecting items in an area with obstacles.

### 4.2.2.1 Terminal Constraints

This simulation had four different ways reaching a terminal state in an episode.

1. The agent made 300 consecutive steps without receiving any reward.
2. The agent attempted to move outside environment grid boundaries.
3. The agent reached a total of 2000 steps.
4. The agent attempted to move against an obstacle.

### 4.2.3 Experiment 3: Cooperative Target Protection

This experiment introduces a different complexity than the previous experiments in the sense that the agent has to adapt to a more tactical behavior while relating to several objects across the environment. The objective of the experiment is to cooperate with another autonomous actor (\textit{actor}_{\text{aut}}) in guarding a target from external autonomous enemies approaching the target from the edges of the environment. The agent can eliminate an enemy by closing in towards it, but that could potentially leave the target unprotected from other enemies coming in from other directions. The number of enemies is constant (see table 4.5), and whenever an enemy is eliminated or reaches the target, a new one spawns at a random location near the edges of the environment grid boundaries.
The agent had a guarding area covering a radius of 10 coordinate units around it. Whenever the target would be within this area, any enemies attempting to reach it would automatically be eliminated. The purpose of the added autonomous actor was to study if the agent could learn how to relate to it in order to optimize its performance in relation to it. Given that the target would be within the guarded area of the agent, the reward of each eliminated enemy would correspond its distance to the autonomous actor (further away - greater reward). This was a way to impose some notion of cooperation between the agent and an autonomous actor, where the cooperative element was solely dependent on the fact that the agent adapted its behavior to the behavior of the autonomous actor.

An extra element of complexity that was added to this simulation was that the autonomous actor could be guarding either the north, south, west or east side of the guarded target. This was chosen at random and was then fixed through the whole episode.

The agent has an edited step length from the general rules of table 4.3 that can be viewed in table 4.3.

<table>
<thead>
<tr>
<th>Edited Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent step length</td>
<td>2 coordinate units</td>
</tr>
<tr>
<td>Number of enemies</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.5: Edited general settings for Experiment 3

This experiment was divided into two sub-experiments, referred to as Experiment 3A and Experiment 3B. In Experiment 3A, the agent can eliminate an enemy, but if its distance from the target is larger or equal to than 10, it will not be rewarded for it. In Experiment 3B, the agent would receive a reward for this action, which adds another layer of complexity by forcing the agent to choose between going for an immediate reward and leave the target unguarded or to guard the target and await the enemies to approach and eventually be eliminated by entering the guarded area of the agent or the autonomous guard. Full details of the possible rewards for Experiment 3A can be viewed in equation (4.3) and the details for Experiment 3B in equation (4.4).

In figure 4.3, an illustrated sequence of the simulation can be viewed.
\[ r_d = \begin{cases} 
  d & \text{distance}_{\text{enemy,agent}} < 10 \text{ and } \text{distance}_{\text{target,agent}} > 10 \\
  -1 & \text{moving outside environment boundaries} \\
  -1 & \text{distance}_{\text{enemy,agent}} < 5 \\
  0 & \text{otherwise} 
\end{cases} \]

where

\[ r_d = \frac{\text{distance}_{\text{actoraut,agent}}}{20} \]

\[ r = \begin{cases} 
  d & \text{distance}_{\text{enemy,agent}} < 10 \text{ and } \text{distance}_{\text{target,agent}} > 10 \\
  0.1 & \text{distance}_{\text{enemy,agent}} < 10 \\
  -1 & \text{moving outside environment boundaries} \\
  -1 & \text{distance}_{\text{enemy,agent}} < 5 \\
  0 & \text{otherwise} 
\end{cases} \]

where

\[ r_d = \frac{\text{distance}_{\text{actoraut,agent}}}{20} \]

Figure 4.3: An illustrated sequence of the third simulation where agent is rewarded by guarding a moving target in cooperation with a programmed actor. The reward is based on the guarded area.

### 4.2.3.1 Terminal Constraints

This simulation had three different ways reaching a terminal state in an episode.

1. The agent made 300 consecutive steps without receiving any reward.
2. The agent attempted to move outside environment grid boundaries.

3. The agent reached a total of 2000 steps.

4.2.4 Experiment 4: Bounding Overwatch

Bounding Overwatch (BO) is a military maneuver performed by two or more soldiers where the purpose is to move towards a given location effectively while advancing as safely as possible through a potentially dangerous area [86]. This maneuver is performed two or more soldiers by advancing indiscriminately, where the stationary soldier guards (or is prepared to take action) while the other one advances.

The objective of this experiment is to train an agent to perform a BO with another manually implemented autonomous actor ($\text{actor}_{aut}$) towards a goal specified. Since this maneuver requires that one actor to move at the time while the other one stands still and guards, an additional guard action was added to signal and trigger the other actor to advance. This action has an effect if and only if it is performed while the distance between the agent and $\text{actor}_{aut}$ is less than 10 coordinate units, which is referred to as the guarded area. When the agent takes that action (and $\text{distance}_{agent,actor_{aut}} < 10$), the agent stands in guarding position while it triggers the $\text{actor}_{aut}$ to autonomously advance to the coordinate point nearest the goal (within the guarded area of the agent). If $\text{actor}_{aut}$ manages to come within 5 units of the goal, the episode terminates, and the agent is rewarded with a reward depending on the length of which the $\text{actor}_{aut}$ advanced. There is no way for the $\text{actor}_{aut}$ to take any actions by itself to reach the goal. The agent is in full control, and the only thing that triggers $\text{actor}_{aut}$ to take action is through the agent taking its guard action. Full details regarding the reward function can be viewed in equation (4.5).

The additions to action size and step length from the general rules of table 4.3 that can be viewed in table 4.6. The illustration in figure 4.4 shows an illustrated sequence of the simulation where the agent initially advances, takes the guard action and how it triggers the $\text{actor}_{aut}$ to advance towards the goal.

<table>
<thead>
<tr>
<th>Edited Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent step length</td>
<td>2 coordinate units</td>
</tr>
<tr>
<td>Agent action size</td>
<td>5 ($\text{Right, Left, Up, Down, Guard}$)</td>
</tr>
</tbody>
</table>

Table 4.6: Edited general settings for Experiment 4
\[ r = \begin{cases} 
    r_d & \text{if action} = \text{guard and distance}_{\text{agent,actor}} < 10 \\
    1 & \text{if distance}_{\text{actor,goal}} < 5 \\
    -1 & \text{moving outside environment boundaries} \\
    0 & \text{otherwise} 
\end{cases} \]

where

\[ r_d = \left( \frac{\text{advanced distance}}{\text{max distance}} \right)^2 \]

and

\[ \text{max distance} = 20 \]

\[ (4.5) \]

Figure 4.4: An illustrated sequence of the fourth simulation where agent is rewarded by advancing towards goal. The reward is also based on advancing within the guarded area and guarding an area, so the programmed actor can advance.

4.2.4.1 Terminal Constraints

This simulation had three different ways reaching a terminal state in an episode.

1. The agent made 300 consecutive steps without receiving any reward.
2. The agent attempted to move outside environment grid boundaries.
3. The agent reached a total of 2000 steps.

4.3 Experiment Evaluation

The evaluation of the models will be performed by studying its ability to learn a rewarding behavior and by studying the fully trained models’ performance (rewards received) in the simulation. In the evaluation process of
Experiment 1-3 (A and B), 100 consecutive episodes of the given experiment will be executed with all terminal states active except #1 (The agent made 300 consecutive steps without receiving any reward). Since one of the objectives of Experiment 4 will be to finish in the least number of steps as well as maximizing the received rewards, it will be evaluated by running 1000 consecutive episodes instead. Both the received rewards and the number of steps per episode will be studied in the evaluation.

As stated in section 1.3, to limit the scope of the thesis, no specific in-depth analysis will be conducted by visually studying the agents’ behavior.
Chapter 5

Results

This chapter presents the results of the experiments as well as the evaluations runs for each experiment and algorithm. In addition to a plot showing the performance of the agent along the training steps, this chapter will also include box plots showing its test performance on a specified number of episodes to visualize the performance between the different algorithms.
5.1 General Overview

The algorithm’s training time varied between the models. But for the sake of easier studying of the training progress, they will be plotted categorically separated into Regular Agent (RA) methods and Asynchronous Agent (AA) methods, since they have been applied with to different max time steps ($T_{max}$ and $T_{gmax}$).

In Table 5.1, an averaged training speed (steps/second) and total training time averaged over each algorithm model can be viewed. All of the models trained with the selected algorithms for these experiments total approximately 400 h (17 days) of computing time. The training time of the models will not be a part of the model performance evaluation but will be taken into consideration as a factor in the discussion section chapter 6.

<table>
<thead>
<tr>
<th>Network Model</th>
<th>Steps/Second</th>
<th>Full Training/Model (≈h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>60</td>
<td>23</td>
</tr>
<tr>
<td>DDQN</td>
<td>55</td>
<td>25</td>
</tr>
<tr>
<td>A3C-FF</td>
<td>1250</td>
<td>11</td>
</tr>
<tr>
<td>A3C-LSTM</td>
<td>800</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 5.1: An averaged summary of training time for each model in each experiment.

Keep in mind that for the RA, the training plot is not fully representative of its actual performance since it, after annealing the exploration factor during the initial phase of the training, kept it fixed at 0.1. However, no exploration was used in the plotted model performance, and are therefore a better suited representation of their final performance. Further, the training plots for both RA and AA are averaged over multiple episodes to enhance the readability of the plots.

5.2 Experiment 1 Results

The progress of both the RA and AA models can be seen in figure 5.1 and figure 5.2.
Figure 5.1: The average reward/episode during the training of the regular agent models of Experiment 1.

Figure 5.2: The average reward/episode during the training of the asynchronous agent models of Experiment 1.

The performance of the agent models can be viewed in figure 5.3.
5.3 Experiment 2 Results

The progress of both the RA and AA models can be seen in figure 5.4 and figure 5.5.

Figure 5.3: The performance of each model in Experiment 1.

Figure 5.4: The average reward/episode during the training of the regular agent models of Experiment 2.
Figure 5.5: The average reward/episode during the training of the asynchronous agent models of Experiment 2.

The performance of the agent models can be viewed in figure 5.6. For this plot, the agents played the simulation for 100 episodes with only terminal constraint 3 active.

Figure 5.6: The performance of each model in Experiment 2.
5.4 Experiment 3 Results

The following experiment result is divided two sections to separate the results of Experiment 3A and Experiment 3B.

5.4.1 Experiment 3A Results

The progress of both the RA and AA models can be seen in figure 5.7 and figure 5.8.

![Figure 5.7: The average reward/episode during the training of the regular agent models of Experiment 3A.](image-url)

Figure 5.7: The average reward/episode during the training of the regular agent models of Experiment 3A.
Figure 5.8: The average reward/episode during the training of the asynchronous agent models of Experiment 3A.

The performance of the agent models can be viewed in figure 5.9.

Figure 5.9: The performance of each model in Experiment 3A.

5.4.2 Experiment 3B Results

The progress of both the RA and AA models can be seen in figure 5.10 and figure 5.11.
Experiment 3B, Regular Agents

Figure 5.10: The average reward/episode during the training of the regular agent models of Experiment 3B.

Experiment 3B, Asynchronous Agents

Figure 5.11: The average reward/episode during the training of the asynchronous agent models of Experiment 3B.

The performance of the agent models can be viewed in figure 5.12.
Figure 5.12: The performance of each model in Experiment 3B.

5.5 Experiment 4 Results

The progress of both the RA and AA models can be seen in figure 5.13 and figure 5.14.

Figure 5.13: The average reward/episode during the training of the regular agent models of Experiment 4.
Figure 5.14: The average reward/episode during the training of the asynchronous agent models of Experiment 4.

The performance of the agent models can be viewed in figure 5.12.

Figure 5.15: The performance of each model in Experiment 4.
Chapter 6

Discussion

In this chapter, various aspects of the experiments are discussed and the final performance of the agents is compared, regarding to which algorithm they were trained with is compared. This chapter also critiques certain aspects of the method used and experiment set up, and gives suggestions to how some parts could have been better and further explored.
6.1 Experiment Discussion

In the following section, the training and performance of all the agents throughout all experiments will be studied. For each experiment, one agent was trained using a Deep Q-Network (DQN), a Dueling Deep Q-Networks (DDQN) and two variants of Asynchronous Advantage Actor-Critic (A3C) (A3C with a Feed Forward Network (A3C-FF) and A3C with an LSTM Network (A3C-LSTM)).

6.1.1 Experiment 1

For the first experiment, all the trained agents seemed to be able to learn to fulfill the objective to some degree. The training progress of the RAs (DQN and DDQN) did appear to have a more unstable learning-process (see figure 5.1) in comparison to the AAs (A3C-FF and A3C-LSTM) (see figure 5.2), that quickly converged towards collecting on average 500 items/episode and then slowly improved along the training steps.

Studying the model evaluation of the agents (see figure 5.3) shows that even though there is a great difference in performance between the two categories, they all had a median above 300 rewards, concluding that they did learn to fulfill the expected objective. One way of explaining the RAs behavior is to say that they appeared "clumsy" or "confused" at times, whereas the AAs were incredibly efficient in the sense that they did not overshoot any of the items - but simply moved in straight paths, just close enough in order to collect the item. An example showing the efficiency of the best performing agent, trained with A3C-LSTM, can be seen in the trace plot in figure 6.1. The plot shows a label above each Item representing which step it was collected on and the circle surrounding it is the minimal required distance to collect it.
6.1.2 Experiment 2

The objective of this experiment was set up to be a bit more complex and difficult than the first experiment. In this experiment, the agent had to move closer to an item in order to collect it (with the required distance now being 5 coordinate units, previously 10). The agent also had to avoid colliding with any of the obstacles in the environment, for which it would receive a negative reward, and the episode would terminate.

The results of this experiment show a noteworthy difference in performance that is not necessarily categorically separable into the categories of RA and AA like in the first experiment. During the training process, the RA seemed to show significantly poorer performance than the AA (compare figure 5.4 to figure 5.5), but the performance evaluation of all agents is not equally separable according to agent type (see figure 5.6). This is potentially an effect of the exploration factor is annealed to 0.1 and then kept there while training the RA, but later set to 0.0 when evaluating the performance of the agent.

The model evaluation also shows that the performance of the A3C-LSTM agent was far superior to the other agent types. The agent using A3C-FF was still the second best performing agent, but performed at worst in level with the RAs.

Both of the RAs also managed to perform the task, but just like in the
previous experiment - visually studying the behavior proved the agent to be less efficient in moving towards the items.

An example showing the efficiency of the (yet again) best performing agents, using A3C-LSTM, can be seen in the trace plot in figure 6.2. The plot shows a label above each Item representing which step it was collected on with no re-spawns, and now with a diametrically half sized circle surrounding it representing the new minimal required distance to collect it.

![Experiment 2, Trace plot](image)

**Figure 6.2:** Trace plot of the best performing agent (A3C-LSTM).

### 6.1.3 Experiment 3

The third experiment had a different kind of complexity to it, where the agent had to learn to fulfill a specific object given other, dynamic, factors and events it had to take into consideration. In experiment 3, an autonomous actor was introduced as well as autonomous enemies that ultimately would impact the performance of the agent. This experiment was divided into two sub-experiments, where the objective ultimately was to guard a given target, but with a two different reward functions. In Experiment 3A, the agent could eliminate an approaching enemy, but would not be rewarded for it. In Experiment 3B, the agent would be rewarded for it - and therefore having to make a decision between approaching the threat to receive that reward (and potentially leaving the target unguarded), or awaiting the approaching enemies (and the rewards for eliminating them) until they reach the guarded...
area within the proximity of the target, in which they would be eliminated. The position of the target and its guard would also alternate at random (between north, south, east and west), adding yet another factor which the agent had to learn to act according to.

Because of this complexity, the hypothesis was that the algorithms in this experiment (especially 3B) would have difficulties converging or performing well. Studying figure 5.7-figure 5.11 shows that all the algorithms in general indeed did seem to have difficulties, but they all managed to learn the objective to some degree - especially A3C-LSTM where its performance again far superior to the other algorithms. The probable cause of this discrepancy could again be the robustness of Long Short-Term Memory (LSTM) cells, where the ability to learn a sequence of states and actions seem to have had an effect on both the performance as well as the convergence speed.

The trace plot in figure 6.3, taken from a randomly selected episode from Experiment 3B, shows the A3C-LSTM agents performance during an episode with the terminal state set to when all enemies were eliminated with no re-spawns. Seen in the illustration is, beside the autonomous actor’s and the guarded targets position, the path of the agent during the episode. Also, beside the path of each enemy, the enemy itself is labeled with which specific step it was eliminated on. The larger circles surrounding the agent and autonomous actors shows their respective guarded area and the overlap between them (which is where the guarded target stands).
Experiment 3B, Trace plot

One difference between the 3A and 3B experiments in comparison to the previous ones is that the addition of a reward based on distance could be considered as an injection of "expert knowledge." Instead of only rewarding with e.g. +1 when a certain objective or goal is fulfilled, the agent receives a reward that is dependent its distance from the autonomous guard (given it is close enough to the target).

The main reasoning behind this is that it was desirable that the agent should learn to keep the guarded area as large as possible. But it is important to keep in mind that injecting expert knowledge should be avoided, as it would mean that training an agent for real military simulations would require actual military domain-expert knowledge to shape it. However, it could be argued that it is necessary in this (or other similar) cases, since shaping the agent behavior beyond just "good", "neutral" and "bad" actions (+1, 0 and -1 in rewards). Having such limits puts far more difficulty on the simulation set up-side and might in the end be more counter-productive than a small injection of expert knowledge after all.

6.1.4 Experiment 4

In final experiment, the agent both had to learn in which situation to take a specific action, but also to learn how to optimize according to that constraint. Since only a small circle (radius 10) would be the actual area in
the environment where the additional action guard would trigger an event (if $\text{distance}_{\text{agent,actor}} < 10$), there was a risk that the agent would have difficulties learning how to fulfill the objective of this experiment simulation. Studying the training progress of the RAs figure 5.13, this seemed to be the case. The RAs performance (see figure 5.15) shows the median of the episode rewards just below, $\approx-0.6$. To see how this compared to taking actions at random, a random agent was evaluated through the same evaluation process, which resulted in a median just above $\approx-0.5$ - meaning that the both the RAs actually performed worse than randomly. This could be an issue that would be solved by an increase in the number of training steps ($T_{\text{max}}$), which experiments by Mnih et al. [20] were set to 10 million (rather than the 5 million in this experiment). However, due to the time consumption of training two new agents with a doubled $T_{\text{max}}$, an extension experiments were considered out of scope for this thesis and will not be conducted.

On the other hand, the AAs did seem to learn how to perform the objective. The biggest difference in the results, comparing to the previous experiments, is that the A3C-FF agent surpassed the A3C-LSTM agent in overall performance, which was contradictory to what one might expect given the earlier results. By studying the figure 5.14, one could see that the A3C-FF converges earlier, but eventually stops to improve from around 25 million steps and onward, while the A3C-LSTM still was improving as the maximum steps was reached and the experiment terminated.

In order to see if A3C-FF would still be superior to A3C-LSTM if the limit of maximum global training steps ($T_{\text{g, max}}$) was increased and due to the lower training time of these agents, two new agents of AAs were trained with $T_{\text{g, max}}$ set to 80 000 000 (80 million) training steps (previously 50 million). All other settings were kept consistently to the original Experiment 4 settings. The result of this extended experiment that can be viewed in figure 6.4 shows that the A3C-LSTM agent indeed managed to surpass the A3C-FF in the extended experiment, even with a slight improvement in the performance of the A3C-FF agent as well. A performance comparison showing the difference between the regular and extended maximum training step can be viewed in figure 6.5.
Figure 6.4: The average reward/episode during the training of the asynchronous agent models in the extended Experiment 4.

Figure 6.5: Performance showing the improvement of the models between maximum training step set to \( T_{\text{max}} \) 50 million training steps and 80 million training steps.

Given these results, the agent trained using A3C-LSTM again proved to be the best algorithm to train the agent with - making it consistently outper-
form all the other algorithms in this thesis. In figure 6.6, a trace plot shows an example of an episode with the A3C-LSTM agent from the extended experiment. The black points mark areas where the agent or the autonomous actor \((\text{actor}_{\text{aut}})\) held a fixed position while waiting for the other one to advance, continuously moving towards the goal area (marked with an x).

![Figure 6.6: Trace plot of the best performing agent (A3C-LSTM).](image)

6.2 Algorithms and Network Architectures

6.2.1 Algorithm Performance

Studying the results in the previous chapter, there is no doubt that A3C-LSTM had the overall best and most robust performance. Though it took slightly longer to train than A3C-FF, it had a better ability of learning complex task where the agent not just had to learn a specific behavior (in the sense of taking a series of actions), but rather to learn a specific tactic. This shows, as explained in section 6.1.3, that it might be more fit in performing more tactically demanding and complex tasks than the other algorithms.

From both the training time and mode performance of the RAs in the experiments, one can conclude that they show little or no advantage in comparison to the AAs. The asynchronous, parallel, agents converge much faster (counting both training time and number of training steps divided
by the number of parallel agents running) and perform on average twice as good (or more).

6.2.2 Networks and Hyperparameter Optimization

A common part of Machine Learning is to use various methods of optimizing hyperparameters for the algorithms with the goal of optimizing the performance of the agent. Studies by Mnih et al. [20][26][21] and Wang et al. [24] have already constructed network architectures and performed hyperparameter optimization for the algorithms used in this thesis, but targeted for e.g. the Atari domain (which has many similarities with the experiment simulations used in this thesis). They found a set of good and robust parameters and network architectures using roughly 5-6 games, and then used them across wide range games where the algorithms still performed well. With that as a basis, and considering the fact of time limitation of this thesis project (as well as the number of algorithms used on the number of experiments), hyperparameter optimization and experimentation with network architectures was considered out of scope due to the long training time of each agent. Instead, the hyperparameters and network architectures used by the studies mentioned above were used as is across all experiments of this thesis as well.

This could mean a number of things. To begin with, all the results should be considered a rough estimate of what could be achieved, meaning that the network architectures and hyperparameters set are anywhere in between the worst possible and best possible settings for the experiments of this thesis project, and that the given experiments are not sufficient to conclude how close to best or worst they are. Potentially, the performance of all the agent could also be improved by experimenting with different architectures and performing hyperparameter optimization.

6.3 Deep Learning Experiment Setup

6.3.1 Reward shaping

When conducting the experiments of this thesis, many ways of shaping the reward function was experimented with in order to have an agent that does not take too long to converge while trying to input as little expert knowledge into the rewards as possible. This proved to be a difficult task, and one must thing carefully when setting the reward function.

The early experiments of this thesis were conducted using a continuous reward function, meaning that the agent would receive a non-zero reward for every state, ranging between negate and positive rewards. This could e.g. be based the distance between the agent and a specific point. One issue this
could cause (depending on the algorithm) was that the agent would adapt a behavior where it would not succeed to fulfill its objective.

Since it is hard to speculate about specific details of why an agent behaves the way it does (due to Deep Learning (DL) and Artificial Neural Network (ANN) in general acting as a black box), one can speculate that a probable cause for this could be that the agent. One hypothesis is that it would learn the bad/non-positive behavior because it would take one action, and since the agent was rewarded for that action, it would be conceived as a good action, and therefore reinforce the possibility of taking the same action for a similar state (by the estimation of the network).

In the final version of the experiments (and their respective simulations), the rewards were set to specific states (hence; not continuous), which led to the agent having a lot less trouble in adapting to a good behavior.

6.3.2 Image Frame Representation

Another issue that was faced while conducting the experiments was that the agent sometimes seemed to fail to learn anything. It did not get worse, it just did not get better either, appearing to act randomly through the entire training process.

A probable cause of this problem was how the environment (and all its components such as the agent, the obstacles, targets etc.) was represented. Initially in early version of the experiments, a very lightweight interface was used where objects were as small as 1x1 pixel in an 80x80 pixels environment that sent as an image frame input to the network. Also, each object only had a single-color representation in gray scale, represented by a value between 0.0-1.0.

All network architectures used in this experiment (see table 4.4) initially had convolutional layers, taking a stack of image frames as an input and serving as an output to the fully connected, deeper, layers. A hypothesis to the non-learning agent was therefore that this issue originated with the image frame input representation and the convolutional layers ability to extract features from it (that would then be passed on to the fully connected layers). If any information could be lost in the convolutional layers due a bad input, the input to the fully connected layers would not make sense to the network either - and the agent will ultimately fail to do anything other than would appear as acting randomly, since it cannot "see" anything.
One of the approaches to solve this issue was solved by using larger representations in all the components of the environment, e.g. 10x10 rather than 1x1. In addition to that, different tones of gray scale were added to each previously monotone object, allowing the Convolutional Neural Network (CNN) to more than one attribute as it convolves the image frames. This gave an immediate (positive) effect in all algorithms and contributed to both a more stable performance increase during training, but also in better performing agents overall.
Chapter 7

Conclusion

This chapter concludes this thesis by looking back at the problem statement and attempting to draw conclusions according to it. It also takes a look at brief look at potential future work related to the subject of this report.
7.1 Using Deep Learning to train Synthetic Actors

One significant aspect to keep in mind with the experiments of this thesis project is that even though the objective might seem intuitive and simple to a human, solving it might be a very complex task for a computer. A human has years of experience and likely many experiences with things about every simulation that reminds us of something else we have seen or learned before. Every model trained could be seen as a brain that prior to the training have had no previous experiences, with the Convolutional Neural Network (CNN) being eyes with no prior stimulus. The agent is never told of the objective, rules or purpose of the simulation beforehand. It does not either know what it controls and how each action will affect the state (at least initially). Eventually, during the training progress convolutional layers learn to ”see” significant objects of the simulation and potentially learn how the agent act in the environment in order to maximize its received rewards.

The main purposes of this thesis project was to evaluate whether Deep Learning (DL) techniques had the potential of performing well learning complex behavior in military simulations like Virtual Battlespace 3 (VBS3). Even though VBS3 was not used in the actual experiments, the simulations used were shaped to prototype a scaled down version VBS3 in the birds-eye view setting. The reasoning behind this was to enable a faster adaption to previous research and improved training speed (since the simulations were much more light-weight and could be better adapted to output an image frame much faster).

Looking at the results of the experiments, one can conclude that the agents could indeed adapt behavior that was beneficial to fulfilling the objective of each simulation; with the Asynchronous Agent (AA) methods were far more superior to the Regular Agent (RA) methods. The agents did also manage to learn to adapt to layered complexity where the decisions made had to be more tactical rather than short-sighted.

Even though the trained agents performed well, it is important to distinguish that each agent was adapted to its specific simulation - meaning that there was no general agent that managed to learn every objective but rather a set of agents learning different behaviors each objective, and doing so well. Using a previously trained model in a new environment with a new objective would performance wise be bad (after all, the agent would be a ”beginner” again) and might also be an issue since the agent is biased from the previous environment. In the best case scenario, the previous experiences of the agent would be overwritten, and it would eventually learn the new task, but potentially also perform poorly on the previous task (basically, overwriting what it had learned).

However, as of right now, the manually implemented actors in VBS3 are also task-specific, and therefore that issue should not be a problem if the simulation agents were trained using DL instead.

76
Assuming that the environment (i.e. VBS3) does not pose an issue in training the agent, using DL methods to train task-specific artificially intelligent agents should certainly be taken into consideration as a replacement to the manually implemented actors. Algorithms like Asynchronous Advantage Actor-Critic (A3C) with an LSTM Network (A3C-LSTM) has previously been proven to be superior and robust in a wide range of applications, and given the result of this thesis - there is good basis to assume it as a good potential to perform well in creating synthetic actors too.

In addition to a good framework is set up for training the agent, the task of producing task-specific agents for simulations using Deep Reinforcement Learning (DRL) would need the following components in order to scale well in full production:

- **A carefully shaped reward function**, which could potentially mean that there would be a need for some expert knowledge in order to shape it. This would also mean that having a domain expert involved would still be needed, but now for the purpose of setting up a reward function rather than shaping a complete behavior model.

- **Computing resources** to train the agents with. Training agents using DL techniques takes a lot of time and computing power and might require specific hardware. However, as the area of DL expands, and new hardware gets better adapted to use it, this issue is slowly being minimized, especially with new algorithms and techniques that are shaped to use less resources (such as Asynchronous Deep Reinforcement Learning (ADRL)).

- **An extensive simulation API**, enabling calls that provide the training algorithms with scaled down and correct visual representations that matches what the actual agent later will take as input (or see) during the simulations. A good example of that is the recent collaboration between Google DeepMind and Blizzard [87].
7.2 Future Work

Due to the recent advancements and successes in Deep Learning research, a growing interest and investment from both research institutes as well as the industry is accelerating the rate of the advancements within the field. Over the course of this thesis project (approximately 5 months), groundbreaking advancements have been made and published in areas like voice synthesis [88], generative models [89] and neural computers [90][91] to name a few, and the progress is showing no signs of slowing down. OpenAI recently released the largest platform of environments for Deep Learning yet, named Universe, as an open source project [92].

With this thesis project, the first step towards generating artificially intelligent actors for a military simulation has been tested and proved successful on a prototype level. It is yet hard to estimate how much more efficient it is than manual implementation, but given the training speed of the agent (once the simulation, reward function and such) is set up correctly, it seems to potentially decrease a large chunk on cost and time otherwise spent on understanding or implementing a behavior model manually.

Progress is also being made in the area where developers are opening up their environments to simplify the training of agents using Deep Learning methods. Recently, Blizzard announced their collaboration with Google DeepMind where an API for StarCraft II is being developed that will make it easier and faster to train agents controlling the game [87].

Such progress with VBS3, as assumed in section 7.1, might not just prove to be useful, but potentially the essential transition from how agents are developed today towards Deep Learning-trained agents.
References


