UTRECHT UNIVERSITY

Deep Learning of Hierarchical Skills for Dynamic Tasks in Minecraft

by

Paolo Bernardo Umberto Luigi de Heer

A thesis submitted in partial fulfilment for the
degrees of Master of Science
ICA-3743683

in
Artificial Intelligence
and
Game & Media Technology
in the
Department of Information and Computing Sciences
Faculty of Science

July 2018

Contents

Chapter 1

Introduction

Currently, many tasks that are easy for humans can be extraordinarily hard for machines, such as driving on the road, recognising objects in unclear pictures, or playing complex computer games. The benefits of enabling computers to perform tasks such as these are endless. For example, if autonomous machines are able to drive perfectly on the road, it could ease congestion and reduce collisions.

The research field of Machine Learning is occupied with, amongst many others, exactly these types of tasks. An example of a learning paradigm from that field is Reinforcement Learning, which from rewards and penalties from the environment, learns the best behaviour from trial and error, in a restricted system [1]. This learning is somewhat similar to how a human learns, by trying various actions, observing the effects, and possibly trying again a different approach.

Recently, a new method coined Deep Reinforcement Learning (DRL) has emerged from the field of Machine Learning, which combines several different Machine Learning paradigms, one of which is Reinforcement Learning. It also uses Neural Networks (NN), a central idea of Machine Learning which will be detailed in section 2.2. DRL proves to be very successful in 'solving' (i.e. learning the optimal behaviour) a wide spectrum of problems, such as for the first time beating the world's best human at the Japanese board game Go [2], teaching robots how to walk efficiently [3] or, using a single learned model, achieving super-human performance in dozens of Atari games [4].

An especially interesting computer game is Minecraft, where a player is free to roam an unbounded computer-generated landscape complete with caves, animals and monsters. It is interesting because it is highly dynamic and adaptable, presenting a real challenge to agents, and can represent our real world. The player has to gather materials to craft items and buildings from scratch in order to survive. As a result of this complex and open-ended nature, Minecraft is exceptionally well-suited for AI research in autonomous behaviour. It has many challenges that are similar to human endeavours such as exploration, exploitation, learning, and cooperation. These challenges are complex and varied enough to pose a promising candidate for developing learning techniques for more general AI, i.e. being able to use a single learning model to efficiently learn generic complex behaviours for different domains.

Typically, the output of a DRL network is a single 'atomic' action. Atomic in this context means an action with the lowest level of complexity, instead of a composite action. In the case of Minecraft it would be similar to the exact command a player would give when playing the game (e.g. move my character forward one unit length). Tessler et al. [5] recently implemented a DRL network in Minecraft, where they extended a DRL network with so-called 'skills' structured in a skill hierarchy, forming a Hierarchical Deep Reinforcement Learning Network (H-DRLN). These skills are encoded in the network as separate subnetworks. Each subnetwork represents a specific type of more complex action, such as 'navigate to a location' or 'pick up an object somewhere'. Tessler et al. show that training the DRL network using these hierarchical skills reaches the desired behaviour more quickly and is able to perform competently on more complex scenarios compared to a 'regular' DRL network, thanks to the prelearnt, decomposed skills.

However, only *static* scenarios and skills in a confined space were used in that study. For example, there were no moving objects in the worlds tested with and no skills were learned that were dependant on other dynamic objects. This more simpler, restricted version of the game world is of course much easier to learn for a neural network since introducing dynamics means an agent's learned model of the world needs to be robust to these changes, which is more difficult. However, it also heavily detracts from the potential of training a *general AI* in the dynamic and intricate world of Minecraft, and by extension the real world.

1.1 Aims

The aim of this thesis is to extend earlier work on Hierarchical Deep Reinforcement Learning to support more realistic problem settings by extending the algorithm to support dynamic obstacles and tasks, and implement a newer and more efficient learning algorithm.

The H-DRLN for Minecraft will be expanded with a Neural Network capable of learning behaviour for more dynamic scenarios, such as avoiding, trapping, or destroying enemies, gathering dynamic items, or interacting with non-player characters (NPCs) to achieve more complicated goals. New skills and skill-subnetworks will be defined that enable learning these new behaviours. After first learning these skill networks independently, they will be combined and the skills of the agent will be tested in dynamic scenarios. The performance of the agent in these dynamic scenarios will be compared to the performance of the H-DRLN trained with the static domains and a non-hierarchical Deep Q-Network (DQN) trained on the dynamic domains. Additionally, a newer, more efficient learning algorithm A3C will be implemented implemented and compared to the DQN algorithm used by Tessler et al. [5]. The results will be a contribution to a more competent and general AI, able to solve more complex and dynamic tasks.

The problems that will be adressed are the following:

1. The H-DRLN relies on some algorithms that have proven to be superseded in some regards. That will be addressed by implementing newer, more efficient algorithms such as QR-DQN and the A2C algorithm. To achieve this, the following will be contributed:
    (a) Implement A2C and QR-DQN next to DQN. A2C and QR-DQN might be more efficient learning algorithms than DQN, and can help make the learning problem more tractable in terms of required computational resources.
    (b) Comparison of the A2C, QR-DQN and DQN learning performance in the static scenarios. This will provide further insight of the performance of different algorithms in complex problem domains.
2. The previously learnt subskills are relatively simplistic, static subskills, which do not adequately represent the complex and dynamic nature of the Minecraft world, and analogously do not translate well to our dynamic real world. To address this, the following will be contributed:
    (a) A learning scenario in Minecraft with a dynamic element will be defined.
    (b) The skills to competently deal with the dynamic element in the new scenario will be developed and learned by an agent. This enables the agent to complete goals with dynamic components in a Minecraft world.
    (c) Learning performance of dynamic and static skills are compared.
3. The H-DRLN architecture is only tested with problems lacking any dynamic components. Insights of the performance on variable domains are missing. That will be solved by designing various domains that incorporate learning dynamic skills and tasks, and testing the performance. To achieve this, the following will be contributed:

(a) Expand the H-DRLN with a dynamic learning component, forming a Dynamic H-DRLN (DH-DRLN). This enables learning behaviour for more realistic and complex scenarios in Minecraft. This gives insight to the virtues and shortcomings of the different paradigms and can further drive the field towards more general AI.

1.2 Outline

First, some background will be given into the development, intuition, and theory of the learning techniques. Next, the most relevant related works and open questions will be highlighted to give an idea of the current frontier of AI research and a motivation for this thesis. Thereafter, the experimental setup and implementation will be detailed. Using this, the results of the tests will be showed and an interpretation of the behaviour and results will be given. Lastly, the findings will be summarised and some insights to future work will be given in the conclusion.

Chapter 2

Background

This section covers the techniques used in Reinforcement Learning, Deep Learning (using Neural Networks), and the relatively novel combination of the two learning paradigms: Deep Reinforcement Learning.

2.1 Reinforcement Learning

Machine Learning can be ’supervised’, where the algorithm can make use of pre-labelled data, or ’unsupvervised’, where the algorithm is not provided with the desired outputs, and has to make a model of the world on its own. Another paradigm within Machine Learning is Reinforcement Learning, an approach stemming from behavioural and game theory research. Reinforcement Learning is somewhere in between supervised and unsupervised, since there is no explicit labelled data given of correct and incorrect outputs, but there is some implicit knowledge passed to the algorithm in the form of rewards (or penalties) from the environment. With Reinforcement Learning, instead of being given a labelled data set, learning is done by finding an optimal policy from trial and error without supervision. The agent does this by repeatedly (in discrete time steps) perceiving a state and choosing an action following from some policy - for every state it encounters. After the action, it perceives a new state and records the resulting reward (or penalty, a negative reward) for ending up in this new state. The reward is used to update its estimate of the expected value of the state and the state-action pair, which is a measure of how desirable the state is to be in. An example of the Reinforcement Learning *perception-action-update loop* is given in the figure 2.1.

While the agent initially knows nothing (or only some heuristic estimate) about the values of states and state-action pairs, through repeated exploration, eventually the

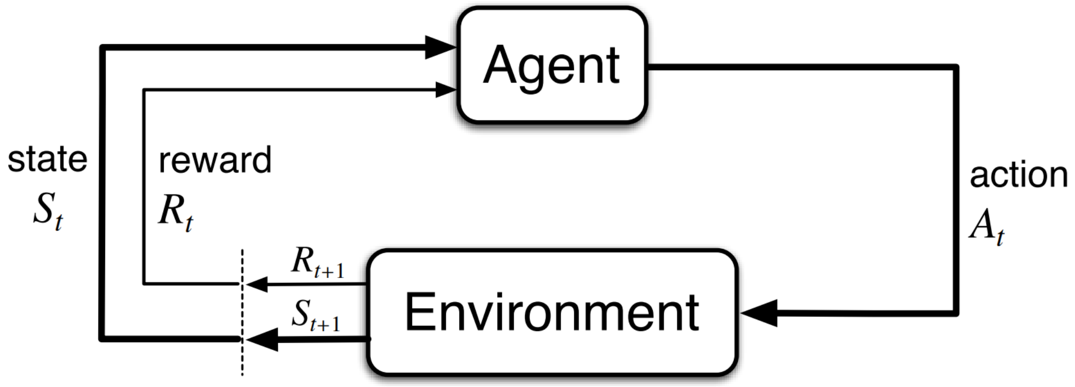


FIGURE 2.1: The general Reinforcement Learning agent-environment interaction loop. Based on the current perceived state, the agent chooses an action to perform. Hereafter, it perceives the new state and reward from the environment. Image adapted from [1].

estimated expected value of all states and state-action pairs should stabilise to the actual expected value, assuming the world (and reward distribution) does not change. When, after enough exploration, the value of all states and state-action pairs has stabilised enough, some new policy can be formed by simply selecting the most promising action for each state-action pair in the set, which gives the most optimal policy to achieve success, given that the initial rewards and penalties were chosen satisfactory.

The first practical research in Reinforcement Learning already stems from the early 90s, when it was mostly used for basic one- or two-dimensional problems such as theoretical grid-world scenarios [6, 7, 8]. However, it also has a much wider practical applicability, such as efficient robotic navigation [3], negotiating economic deals or network traffic [9], simulating emergence of social conventions in groups [10] and intelligently playing complex computer games [4].

2.1.1 Framework

The Reinforcement Learning framework comprises an *agent* that perceives a *state* s_t from an *environment* at timestep t. The agent interacts with the environment by taking an *action* a_t in state s_t. After taking such an action, the environment transitions to a new state s_{t+1} based on the current state and the action executed by the agent. The state should contain all relevant features of the environment and should, together with the chosen action, be enough to determine the next state, satisfying the Markov property [1].

The optimal sequence of actions the agent can take arises from the *reward distribution* in the environment. After taking an action, the agent receives a scalar reward r_{t+1} from the environment. The purpose of the agent if to learn a *policy* π that maximises the cumulative, discounted reward, called the *expected return*. The policy is a function mapping a state to an action distribution, which can be stochastic. An optimal policy is any policy that maximises the expected return in an environment.

2.1.2 Markov Decision Process

Defined as a Markov decision process (MDP) [11, 12], Reinforcement Learning consists of:

- A set of states $\mathcal{S}$, including a subset of starting states.
- A set of actions $\mathcal{A}$
- A transition function $\mathcal{T}(s_{t+1}|s+t, a_t)$ mapping state-action pairs at time t to a distribution of states at time $t+1$.
- A reward function $\mathcal{R}(s_t, a_t, s_{t+1})$.
- The discount factor $\gamma \in [0, 1]$ where a lower value gives more weight to more immediate rewards.

The policy π maps states to a probability distribution over actions: $\pi : \mathcal{S} \rightarrow p(\mathcal{A} = a|\mathcal{S})$. In an *episodic* setting, the state is reset after T timesteps. One such run is called an episode and the sequence of states, actions and rewards constitutes a *trajectory* of the policy. Every trajectory results in the return $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$. The goal of Reinforcement learning is to find an optimal policy π^* such that it achieves the maximum expected return over an entire trajectory: $\pi^* = \underset{\pi}{\arg\max}\mathbb{E}[R|\pi]$. For non-episodic MDPs where $T = \inf$ setting $\gamma < 1$ prevents a diverging sum.

The Markov property, stating that a transition to s_{t+1} only depends on the past state s and action a, is often unrealistic since the environment is usually not fully observable. It is often generalised to partially observable MPDs (POMDPs), where the agent observes what it believes is the environment from the current state and previous action. A recurrent neural network (RNNs) is usually used to formulate the observation of the environment.

2.1.3 Value Functions & the Bellman Equation

Solving a Reinforcement Learning problem is done by iteratively updating the estimated values, using a value function. The following *state-value function* $V^{\pi}(s)$ gives the expected return given by starting in state s and following policy π hereafter:

$$V^\pi(s) = \mathcal{E}[R|s, \pi] \tag{2.1}$$

Translated to the optimal policy π^* gives the optimal state-value function:

$$V^*(s) = \arg\max_\pi V^\pi(s) \quad \forall s \in \mathcal{S} \tag{2.2}$$

If the optimal state-value function was known, the optimal policy is simply selecting from all actions at s_t the one that maximises the expected return. However, since that is unavailable, an extra function is used that gives the *state-action value* called the *quality function* $Q^\pi(s, a)$ which is analogous to V^π with the first action already decided:

$$Q^\pi(s, a) = \mathcal{E}[R|s, a, \pi] \tag{2.3}$$

Given this quality function, the optimal policy can be retrieved by choosing the next action greedily at every state: $\arg\max_a Q^\pi(s, a)$. $V^\pi(s)$ can then also be expressed as $V^\pi(s) = \max_a Q^\pi(s, a)$.

To actually learn Q^π, the function is defined as a Bellman equation [13]:

$$Q^\pi(s_t, a_t) = \mathcal{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))] \tag{2.4}$$

Using Dynamic Programming and the recursive nature of this function, the estimate of Q^π can be iteratively improved by bootstrapping with the current estimate of Q^π, as used by the Q-learning [14] and the SARSA algorithms [15]:

$$Q'(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha\delta, \tag{2.5}$$

where α is the learning rate and $\delta = Y - Q^\pi(s_t, a_t)$ is the temporal difference (TD) error. Y is a target as in a regression problem. Since SARSA is on-policy, it seeks to directly improve the active policy Q^π so that $Y = r_t + \gamma * Q^\pi(s_{t+1}, a_{t+1})$. In contrast, Q-learning is off-policy, meaning that it updates the current active policy using the most promising transition value, resulting in $Y = r_t +]gamma \max_a Q^\pi(s_{t+1}, a)$ which directly approximates Q^*.

2.1.4 Monte Carlo

Rather than using dynamic programming to improve the value functions, Monte Carlo methods can be used to estimate the expected return of a state by averaging the return from multiple runs of a policy [16]. However, this method can only be used in episodic MDPs, since the function needs to terminate to calculate the return of that run. Monte Carlo methods and TD learning can also be combined as in the TDλ algorithm [1]. The λ here defines an interpolation between Monte Carlo and bootstrapping.

Another method learns the *advantage* function $A^{\pi}(s, a)$ [17]. Instead of giving an absolute state-action value function Q^{π}, A^{π} uses relative state-action values, defined as follows:

$$A^{\pi} = V^{\pi} - Q^{\pi} \tag{2.6}$$

Learning relative values removes the baseline value, and can be thought of as that it is easier to learn that one action is better than another than it is to learn the actual returns for all actions.

2.1.5 Policy methods

Instead of relying on value functions to learn the optimal policy, Policy methods aim to directly learn an optimal policy by means of a parametrised policy π_{θ}. The parameters are updated, usually using a gradient-based method in Deep Reinforcement Learning. Gradients can give an effective learning signal as to how to update the policy parametrisation.

To compute the expected return, trajectories of the current policy parametrisation are averaged using a sampling method, such as Monte Carlo sampling. An estimation of the gradient is needed in the form of the REINFORCE rule [18]. This rule increases the selection probability of the sampled action weighted by the return. However, this computation relies on the empirical return of policy run, and has a high variance as a result. One way reduce this variance, the average return can be taken over several episodes.

Value functions can be combined with this form of policy parametrisation in *actor-critic methods* [19]. Here, the actor learns by using feedback from the critic, using two seperate policies. This way, the methods mediate between variance reduction from policy gradients and bias introduction from value functions.

2.1.6 Hierarchical Reinforcement Learning

Another way to combat the curse of dimensionality is by introducing a Hierarchy in the learning structure, where the complex task is recursively divided into smaller subtasks, creating a task tree. This is actually quite similar to how humans would break down a complex task into smaller simpler ones, and solve those in succession. When using a task hierarchy, the agent first learns the simpler (lower level) tasks. When those are learnt, it starts learning the next, higher level of tasks, all the way up to the root of the hierarchy. This way, the agent can filter out variables that are irrelevant to the current subtask. As a result, many different states are abstracted to a single state for the purpose of that subtask, and as such are much quicker to learn. A second way this hierarchy benefits the learning process is that the same lower-level subtask can be used by multiple different higher-level tasks, further relieving the need to learn the same functionality in different settings, supporting the transfer learning capabilities of the agent. Some research has already been done in this area, such as MAX-Q [20] and Options [21]. Research has shown Options to be capable of speeding up convergence of RL agents both in theoretical [21, 22] and practical approaches [23, 24].

2.1.6.1 Options

The concept of Options within Reinforcement Learning was first developed by Sutton, Precup and Sing in 1999 [21]. It extends the Reinforcement Learning framework by allowing a policy to not only select *primitive actions*, but also invoke other policies, also called options, or (sub)tasks. These options are temporally extended actions (TEAs), meaning a selected option continues executing its policy for multiple timesteps. With this framework, a policy *hierarchy* can be constructed, where the policies closer to the root focus on higher-level goals, and sub-policies deeper in the tree deal with more specialised directives. Usually these sub-policies and hierarchy are defined in advance by a designer [25, 26]. However, research is also being done in generating these options from the data itself, which remains an open research problem [27, 28, 29]. Usually these sub-policies are first learnt individually, in specialised environments to help learn that specific skill. Afterwards, these options are combined in the task hierarchy for use in the actual, more complex tasks.

2.1.7 Limitations

However, Reinforcement Learning also suffers from some glaring limitations, as detailed below. These issues unfortunately still greatly restrain the practical applicability of

Reinforcement Learning in more complex settings.

2.1.7.1 Curse of Dimensionality

Due to the nature of Reinforcement Learning, every extra variable in an environment constitutes a new dimension in the set of states, meaning that as the problem task grows more complex, the state space grows exponentially, and with it the time it takes to learn an optimal policy. This is coined the curse of dimensionality by Richard E. Bellman [30], and is one of the major obstacles of Reinforcement Learning. This is a significant problem for single-agent systems, and doubly so for multi-agent systems (MASs). This is because not only does the state space grow exponentially for every variable that exists, but it also adds that many dimensions for every extra agent introduced to the system, further constituting a significant increase in computing power required to find the optimal policies.

Another difficulty of applying Reinforcement Learning to multi-agent systems or dynamically changing environments is the fact that multiple actors now perceive and influence the world instead of a single one. Instead of a single actor influencing the world and perceiving the result of its own actions, with a multi-agent system, there are multiple actors influencing and changing the very same world. Multiple agents can either act in parallel, or in sequence, and as they change the world, what they previously learnt about the world might no longer be true. As every agent tries to learn the optimal policy, they also change the world for others, requiring others to adapt, which might change what the optimal policy is, requiring others to adapt again, and so on. This makes it more difficult to converge to a global optimal policy, but bears interesting resemblances to human social interactions [31, 32]. This effect is present in different ways in different scenarios, such as collaborative versus competitive agents, global learning versus individual learning, with or without (possibly noisy) communication, etc. These problems are analogous for dynamic environments.

Due to the fact that it can take an agent thousands of episodes (an episode is a single, complete 'play' through a problem) to learn and converge to an optimal policy, the computational needs are excessively high. It is thus severely constrained by the available processing power of the computers at the time of research and as a result, it was an infeasible method for most practical tasks when Reinforcement Learning was initially developed.

These limitations often invoke the need for multiple assumptions and simplifications, for example concerning the state abstraction and the possibility of communicating between agents (for MASs), especially when providing guarantees on convergence and optimality.

2.1.7.2 Need for a Human Designer

Another issue is the design effort required. In addition to the parameters that need to be set, there are also some very domain dependant variable elements in RL: the state abstraction, reward distribution and - in the case of Hierarchical RL - the skill hierarchy all need to be defined by a human designer. Such a designer is required to have sufficient domain knowledge to efficiently encapsulate all relevant features, rewards and recursively decompose the overall task into a collection of subtasks. Having a human design the hierarchy can quickly become infeasible or incomprehensible for more complex task structures.

2.1.7.3 Tabular Storage

Additionally, classical reinforcement learning stores Q-values in a tabular fashion. For more intricate and complex domains, such an implementation would need unrealistically large tables to store all the Q-values. For example, to store a value for each possible combination of pixel brightness values for even a very small computer screen would take more entries in the table than there are atoms in the universe. Even though the space required can be optimised for some settings, there are many environments where either further optimisation becomes infeasible or the initial space required to store the tables is already impossibly big.

However, thanks to significant improvements of processing power in recent years, more complex problems can be learnt. As a result, RL is undergoing something of a revival in terms of popularity and feasibility. Learning the optimal policy for complex tasks that seemed nigh unsolvable before, suddenly enters the realms of possibility. This increase in computing power also enables more MAS applications of RL. As a result of this increase in power, there is also a resurgence of research being conducted on the various forms of RL, further establishing this learning paradigm as one of the most interesting and valuable learning algorithms. Still, the part where feasibility of RL consistently falls short is on the fact that the states are stored in a tabular way, still accompanied by the curse of dimensionality.

2.2 Deep Learning

A different paradigm of Machine Learning is 'Supervised learning', where an algorithm is designed to predict the value of certain properties of an input item [33]. To learn the correct output, it has access to a data set that is already classified, i.e. for every data point, the desired resulting output is known for the algorithm. The purpose is to be able to generalise to unseen data points, predicting the (now unknown) desired output as correctly as possible.

This type of Machine Learning can be used for image classification, patient diagnostics, predictions of stock behaviour based on past events, etc. The output can range from a codified single value (e.g. the expected value of a property), to a range of values (e.g. the certainty that a picture is of a horse, a house, or a human, respectively).

The input and desired outputs are codified as a single value or a multidimensional vector of values, called a tensor, where every value of the vector is a feature. As such, a relatively straightforward polynomial can be specified that calculates a specific output directly dependent on the inputs. This polynomial manipulates each feature with a separate weight parameter, and adds a bias parameter.

To measure how accurate such a learning model is for a specific input or data point, the error between the output resulting from the function, and the 'real' value that was given for that data point is calculated. Usually the average error is calculated for a lot of data points at the same time for more accurate results. The global task of deep learning is to minimise this average error, specifically for the test data. Since these calculations are performed on a lot of data points using the same parameters at the same time, the data can be structured in multidimensional vectors and matrices, resulting in more efficient computation and solving of the task.

In Deep Learning, each unit processes the input using its so called activation function, where unique weight and bias parameters influence the output. This output is forwarded to the next layer of units, until the end. Using training data, it adapts the activation functions to eventually produce correct responses to as many of the (training) input data as possible. It is a very powerful tool to help learn and parametrize.

2.2.1 Neural Networks

Neural networks are a group of non-linear, parametric learning functions. They are called networks since they are a collection of functions (nodes) that together form an acyclic graph. They are hierarchical in nature, in the sense that it internally first learns to discern small, atomic, features, and using a composite of those atomic features to iteratively discern more complex features. Given a dataset $\mathcal{D}$, a neural network's purpose is to find the optimal parameters θ^* that minimizes a *loss function* [33]. The network is separated into layers, where each layer represents a computation of the form:

$$h_1 = f_1(W_1 \cdot x + b_1) \tag{2.7}$$

Where f_1 maps the multidimensional input x of the model to the *hidden unit* - or the next layer - h_1 using weights $W_1 \in \theta$ and biases $b_1 \in \theta$. f_1 is called an *activation function*. The output (of this function) of one layer can be the input (x) for another layer, forming the hierarchical aspect of neural networks. More layers enable to learn more complex features, and as such Deep Learning methods often employ networks with a large number of layers. However, with more layers, parameters, and higher-dimensional input features, the model becomes harder to train. Much current research focusses on methods that are capable of efficiently training deep neural networks.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are able to process sequential data. To make use of information incorporated in a sequence, such as data changing over time, an RNN uses a *memory* to understand the *context* of the data. Using *hidden states*, information from the past is passed through the network [33]. The RNN function is as follows:

$$h_t = f(x_t, h_{t-1}|\theta) \tag{2.8}$$

Here, x_t and θ_t are the input and parameters at this timestep. h_{t-1} is the hidden state of the network at the previous timestep. The parameters θ are shared over all timesteps, and assists in generalising to variable sequence lengths.

An important feature of an RNN is *backpropagation through time*. Since the result at time t depend on all previous timesteps, the loss, or error, from this result needs to be propagated back through all these timesteps. For an episodic (finite) task, it could be propagated through all steps. However, it is computationally expensive to go very far back in time. Instead, the number of timesteps it propagates back through is set to a finite number τ.

One version of a RNN is the *Long Short-Term Memory Network* (LSTM). While most basic RNNs have difficulties with long-term dependencies in a sequence, the LSTM are able to learn these long-term relations without overlooking the short-term dependencies and have proven very successful in various domains [34]. The main component of a LSTM is the *cell state* C_t, which is sparsely updated and can be seen as a separate 'long-term memory'. It also still employs a short-term memory in the form of h_t. Given input x_t and hidden state h_{t-1}, the cell state C_{t-1} is updated to C_t. The new cell state is then combined with x_t and h_{t-1} to return the new output h_t.

2.2.3 Optimisation Algorithms

As stated, to actually *learn* with a neural network some loss or error should be minimized. This is done using an optimisation algorithm. The most common used optimisation algorithm is **gradient descent** [33]:

$$\theta^{t+1} = \theta^t + \alpha \nabla_\theta \mathcal{L} \tag{2.9}$$

Here, $\nabla_\theta \mathcal{L}$ is the vector containing all partial derivatives, called the gradient of $\mathcal{L}$, with respect to θ. Intuitively, this updates the parameters by computing the direction to move the function in. Since the direction is based on the partial derivatives, it can always be chosen such that it moves downhill (i.e. closer to a minimum), given that the learning rate α is chosen sufficiently small as not to overshoot the minimum. Usually, to speed up learning and avoid indefinite overshooting, the learning rate starts out relatively large and shrinks over time to ensure an accurate approach to the minimum. The computation of the gradients through the neural network is called *backpropagation*.

Stochastic gradient descent is a common adaptation where the gradients are updated using only parts of the dataset to reduce the computational costs. This is done by stochastically sampling so called *mini-batches* from the dataset $\mathcal{D}$. The parameters are updated as follows:

$$\theta^{t+1} = \theta^t + \frac{\alpha}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \mathcal{L}_i \tag{2.10}$$

Where $\mathcal{B} \in \mathcal{D}$ is a mini-batch. The size of the mini-batches determines the variance of the gradients: setting it too small might steer it in the wrong direction due to skewed samples, but setting it too large might prevent getting out of a local minimum.

2.2.4 Limitations

While neural networks, both shallow and deep, have already proven their practical effectiveness long ago, there are still some limitations inherent in this paradigm of learning.

Most importantly, neural networks require a lot of training data, which also need to be pre-annotated by humans. This is a heavy reliance on supervision that is not practical. It would be beneficial to progress towards more unsupervised learning and to be able to do away with the necessity for annotation data altogether. Combining reinforcement learning with neural networks enables just that, if at least in part. The input data no longer needs to be annotated, as the rewards - which may be received delayed in time - provide feedback necessary for learning.

2.3 Deep Reinforcement Learning

Most recently, research has been focused on combining the novel Deep Learning techniques with the well-established Reinforcement Learning to constitute so-called Deep Reinforcement Learning (DRL) [35].

Also, the mentioned effect of the curse of dimensionality in reinforcement learning is such that the basic, tabular way to structure the values of state-action pairs becomes unusable: the number of states - and subsequently the size of the tables - simply becomes too large to store digitally. This is another way where the Deep Learning paradigm can help solve the problem. Instead of saving the Q-values in a tabular structure, the values are stored inherent in the structure and functions of the many neurons of the network. This helps to make learning over very complex states and inputs feasible.

In DRL, the basic learning loop is still the same as in reinforcement learning: based on a received state, an agent uses a policy to choose an action to perform (based on the Q-values of the possibilities), after which it receives a new state and reward. That new reward is used to update the current Q-value of the new state, using the Q-function. The major difference is that now, a neural network is used to represent the action value (Q-value) function. Instead of finding the Q-value in a table $Q(s, a)$, the network encodes the Q-values in a non-linear function $Q(s, a|\theta)$ where θ are the weights and biases of the neurons, the parameters of the network. Also, instead of using the value-iteration update rule of Q-learning, every learning update now changes the parameters θ according to the chosen error minimization function, such as gradient descent.

The pioneering work in DRL was prone to unstable or even diverging learning behaviour when using non-linear function approximators such as neural networks to represent the Q-function. In addition, the learning process could take too long, requiring very many samples. Recent work such as the Deep Q-Network detailed below have been able to produce much more stable parametrisations that can also learn much quicker using fewer samples.

2.3.1 Deep Q-Network

One of the most recent big breakthrough Deep Reinforcement Learning algorithms was the Deep Q-Network (DQN) [4, 36]. It was able to play a large number of Atari 2600 video games from the Arcade Learning Environment (ALE) [37] on par with a professional video game player, with only a single model. The structure of the DQN can be seen in Image 2.2. As input, it takes the latest four greyscale frames from the game. These frames are processed by several convolution layers that extract spatiotemporal features, such as movement of the ball or enemies. Next, multiple fully connected layers process the feature map that more explicitly encode the effects of the actions. The final fully connected layer outputs $Q^{\pi}(s,a)$ for all action values in a discrete set of actions. This allows the best action $\arg\max_{a} Q^{\pi}(s,a)$ to be chosen from only a single forward pass through the network, and allows better encoding of action-independent knowledge in the lower, convolutional layers.

The number of the states needed to represent this in a tabular storage would be $|\mathcal{A}| \times |\mathcal{S}| = 18 \times 256^{3\times 210\times 160}$. It would be impossible to create a table of that size, and even if it was, it would be sparsely updated.

The DQN is used as a function approximation for the Q-function, similar to regular Q-learning. The parameters θ of the network can be trained using gradient descent by minimizing the *Mean Squared Error* (MSE). For Q-learning, this is the TD-error:

$$MSE(\theta) = \mathcal{E}_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a'|\theta) - Q(s, a|\theta))^2] \tag{2.11}$$

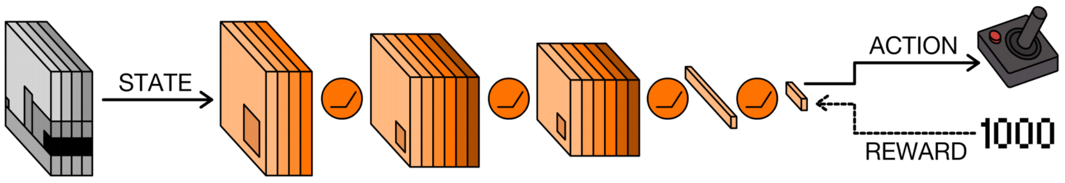


FIGURE 2.2: The DQN gets the state as a set of greyscale frames and processes it with convolutional and fully connected layers. The final layer of the network outputs a discrete action that corresponds to one of the possible inputs for the game. Based on the current state and chosen action, the game returns a new score. The DQN interprets the reward as the difference between the new score and the previous one. This is done by updating the estimates of Q, and the error between the previous estimate and the new estimate is backpropagated through the network. Image adapted from [35].

While neural networks often induce some instability, Mnih et al. [4] introduced two techniques to combat this issue: experience replay and target networks.

Experience replay stores transitions as tuples of $(s_t, a_t, s_{1+1}, r_{t+1})$ in a cyclic buffer, which can be sampled from to train the network with previously observed data [38]. The benefits are that it can significantly reduce the amount of interactions needed, and experiences can be sampled in batches to reduce the variance of the learning updates. Stochastically sampling also reduces the effect of temporal correlations that can hinder the learning process. Lastly, sampling batches can be more efficiently processed thanks to parallel computations on the data, increasing the throughput. An extension of the experience replay memory is *prioritized replay*. Using prioritization, instead of uniformly sampling from the experience replay, trajectories with more important transitions (a higher priority) are replayed more frequently. This results in more efficient learning, and is also applied in the DQN [39].

A **frozen target network** is the other technique used with the DQN. This network helps combat the instability arising from updating the parameters θ at every timestep, which affects the entire network and can lead to oscillations or obstructions in learning. To prevent this, Mnih et al. introduce a secondary network that initially contains the weights of the network enacting the policy, but it is kept frozen for a large number of timesteps. Instead of having to calculate the TD error based on the heavily fluctuating estimates of the Q-values, the policy network uses this fixed target network. During training, the weights of the target network are only updated to match the policy network once per a set amount of timesteps, called the *freeze interval*.

The algorithm for the DQN including these adaptions can be seen in Algorithm 1.

Algorithm 1 The Deep Q-Network with Experience replay and a Frozen target network

```
 1: Initialize replay memory $D$ with capacity $N$.
 2: Initialize action-value function parameters $\theta_0$ with random weights.
 3: Set target action-value function parameters $\theta^- \leftarrow \theta_0$.
 4: $i \Leftarrow 0$
 5: while $i <$ max iterations do $s \leftarrow s_0$
 6:     while $s \neq$ terminal do
 7:         With probability $\epsilon$ select random action $a$
 8:         Otherwise $a = \arg\max_{a'} Q(s, a'|\theta_i)$
 9:         Take action $a$
10:         Observe next state and reward $(s', r)$
11:         Store transition $(s, a, r, s')$ in $D$
12:         $s \leftarrow s'$
13:         if $i \% freezeinterval == 0$ then
14:             $\theta^- \leftarrow \theta_i$
15:         end if
16:         $B = (s_j, a_j, r_j, s'_j)_{j=1}^{batchsize} \subseteq D$
17:         for each $(s_j, a_j, r_j, s'_j) \in B$ do
18:             $y_j = r_j + \gamma \max_{a'} Q(s'_j, a'|\theta_-)$
19:             $\mathcal{L}_j = (y_i - Q(s_i, a_i|\theta_i))^2$
20:         end for
21:         $\theta_{i+1} = \theta_i + \frac{\alpha}{|B|} \sum_{j=1}^{batchsize} \nabla_{\theta_i} \mathcal{L}_j$
22:         $i = i + 1$
23:     end while
24: end while
```

Double DQN is an adaptation of DQN that prevents overly optimistic estimates of the value function. This is done by executing the action selection with the current network θ and evaluating the action with the target network [40].

2.3.2 (Asynchronous) Advantage Actor-Critic

A significant recent improvement over DQN comes in the form of the asynchronous advantage actor-critic (A3C) algorithm [41]. A3C combines advantage updates with the actor-critic method, and uses asynchronously updated policy and value function networks trained in parallel with multiple processing threads. Training using multiple agents in their own separate environments not only stabilises parameter updates, but also allows for more exploration. A3C is used a lot in current DRL projects and is also being adapted and improved. One such adaptation is the inclusion of experience replay in the algorithm [42].

More recently, a synchronous, deterministic variant of A3C has been introduced coined A2C [43]. A2C waits for each agent to finish its episode before updating the policy and value function network. The benefit of this variant is that it can more effectively make use of GPUs parallel computation capabilities, especially benefiting from large batch sizes. A2C is reported to be more cost-effective than A3C when using single-GPU machines, and to be faster than a CPU-only A3C implementation [44].

2.4 Hierarchical Deep Reinforcement Learning Network

Recently, Tessler et al. were one of the first to implement a DRL network with a hierarchical structure [5]. Their goal is to develop an agent capable of lifelong learning - using knowledge transfer - in a complex game environment such as Minecraft. Their architecture uses a modified version of the DQN, extended with the Options framework for RL. It was implemented and tested in Malmo, Microsoft's AI research platform built with Minecraft [45].

The overarching architecture is the Hierarchical Deep Reinforcement Learning Network (H-DRLN). This is a deep neural network, based on a modified DQN, that as input receives downscaled pixel images from Minecraft, and outputs either one of six atomic actions (move forward, rotate left or right, pick up or drop an item, and break a block) or the identifier of a specific subskill. If it selects a specific subskill, it forwards control of the agent's inputs and outputs for $K = 5$ steps to the neural network of that specific subskill. After that number of steps, control returns to the H-DRLN to select the next action(s).

The agent first independently learns four different subskills (navigating, picking objects up, destroying blocks, placing objects) in separate scenarios. The skills are learnt using separate Deep Skill Networks (DSNs), which is mostly like a DQN. After learning, the DSNs managed to solve their sub-domains in nearly 100% of the episodes.

These DSNs are then combined in their H-DRLN architecture, by means of either a DSN Array, or a Multi-Skill Distillation (MSD). The MSD is a novel technique developed by Tessler et al. It is a single neural network that combines the features of the different skill DQNs into one. It performs slightly better than a DQN Array and can scale better to more skills since it is a single distilled network.

The H-DRLN is first tested with a single skill (navigation) in a different, more complex scenario than what the skill was trained with. Here, it performs better than both a vanilla DQN, and the individual DSN. Next, the H-DRLN is tested with a more complex scenario where it must combine all the subskills it has previously learnt. This scenario was tested with a baseline double DQN (DDQN) network, a H-DRLN with DQN and DSN array, a H-DRLN with a DDQN and DSN array, and a H-DRLN with a DDQN and a MSD. The simple baseline DDQN was unable to solve the task after more than 240 epochs. One epoch here means the one cycle of training for some amount of steps or episodes, and evaluating (testing) for some other amount of steps or episodes. Of the H-DRLN implementations, the H-DRLN with a DDQN and the MSD performed best, scoring a 94% success rate and variance of 4%, compared to 10% for the simple H-DRLN.

It shows the effectiveness of distilling a task into subtasks, with related subskills to learn, enabling ever more complex behaviour to be learnt. However, skills have to do with relatively simple tasks, as they remain static. This architecture will be extended to deal with tasks that have dynamic elements, such as avoiding and defeating monsters.

2.5 Summary & Objectives

Each of the techniques shown above provide features that are beneficial - some even critical - to creating an intricate and general learning system. Reinforcement Learning provides the basis in the form of the action selecting policy and the learning algorithm, which continually updates and improves the policy while exploring the environment. The (deep) Neural Network provides a way to store the state space, resulting in a more dense and generic abstraction. The effects of this are that the state space can be of a much higher dimension, that the transfer learning ability is enhanced and that fewer training steps are usually needed, compared to regular RL. Using a deep neural network allows more complicated patterns and connections to be learned in the input data. The learning algorithm provided by the Reinforcement Learning component is the driving factor changing the weights and biases of the neural network. Lastly, training skills hierarchically - where an agent trains using multiple separately pre-trained subskills - can significantly increase learning performance for complicated tasks.

By combining these techniques in a Hierarchical Deep Reinforcement Learning network, it is possible to train an agent in more detailed 3D environments, using more complex and generic tasks. This can lead to more effective use of these techniques in real-world situations, such as robotic exploration and assistance in disaster areas, or agents that have to be able to do a multitude of different tasks. Due to the generic nature of the examined algorithms, these techniques can provide a stepping stone towards more general AI.

However, one limitation of this architecture is that learning can still take a prohibitively long time to learn a given task. Since the previous research, multiple newer deep reinforcement learning algorithms have been developed, that boast better performance. These might severely shorten the amount of learning required for adequate results. Additionally, one of the aspects not fully explored in the original research is that no skills or tasks bore dynamic elements. This is often encountered in real-world tasks, and is expected to possibly further complicate learning. As such, it is highly relevant to research these types of tasks if deep reinforcement learning is ever to be used in real-world tasks as part of a general AI system.

Therefore, the scope of this study is to explore the effectiveness of these methods in the static and dynamic domain in Minecraft. Specifically, some parameter settings will be explored, new algorithms will be compared, and lastly the static subskills will be compared to a dynamic subskill. These experiments are detailed in the next chapter.

Chapter 3

Methodology

Since this study aims to contribute to more capable and general AI agents that can operate in our complicated real world, several experiments will be done to explore the performance of state-of-the-art deep reinforcement learning algorithms in a three-dimensional environment, learning static, dynamic and hierarchical skills. In order to study this, a set-up similar to H-DRLN was used [5].

The goal here is to explore the efficacy of the methods in more dynamic environments in order to see how well they might scale to more complex settings. The rationale behind this is that simulating a world in Minecraft, and having an agent train from pixels as input, is not all too different from a robot interacting in our real world, and testing these improvements will show how suited these methods are to operate in our world. To this end, three different experiments will be performed. The first tests new Deep Reinforcement Learning algorithms that have not yet been performed in a Minecraft-like three-dimensional environment. The second experiment tests five subskills, including the dynamic task, and the third experiment tests a H-DRLN that uses the trained dynamic task. These experiments provide new insights as to the performance of those algorithms in environments more like our own complex world, instead of 2-dimensional arcade games, and how well they perform on more complicated, dynamic tasks.

The experiments are implemented using Malmo, Microsoft's AI research project built on Minecraft [45]. The Malmo framework can provide direct state information from the game to the agent as input to the neural network, and the agent can give direct actions to perform back to the game. Part of Tessler's code is used for the domain-, agent- and learning algorithm-interactions [46]. To construct the neural networks, the deep learning library PyTorch is used [47]. The resulting code is hosted on the following github repository: `https://github.com/Phantomb/malmo_rl`.

The previous research used a different Minecraft research platform named BurlapCraft [48], which has been discontinued. There are some substantial differences in the platforms, most notably leading to a different form of action execution in Malmo. As a result, the agent's available actions in this study differ slightly from the original experiments (the agent can only turn 90° instead of 30°), possibly leading to differing results in learning efficiency and efficacy. This is an implementation difference that is unfortunately unavoidable, and will be taken into account when discussing the results.

In spite of the limitation stated above, the intention is to deviate as little as possible from the original research. Staying as close as possible to the source is important, since every alteration can have severe impact on the performance of the agent. In this pursuit, all parameters are modelled after the values as stated in the original research in [5]. Unfortunately, not all parameters are (unambiguously) stated in the paper, nor do the ones that are stated provide the expected results in preliminary exploratory tests. In these preliminary tests performed on the relatively simple single room domain for example, the agent was not able to reach a stable success rate above 50%. Because of this, a parameter exploration is performed in order to find a set of parameters and settings that provides positive and workable results. This provides a significant hurdle to the experiment, because it constitutes further deviation from the original, hindering the ability to compare results.

Additionally, due to the way the Minecraft platform is coded, the number of steps that can be taken per second is relatively limited. On average, an agent made in this implementation, using a high-end PC, executes 2-3 steps per second. That means that, for a test running for 60 epochs with the settings as described below, it takes around four days of simulation. This is another prohibiting factor in performing an full exhaustive search for the best parameters. Therefore, some most likely candidates for the most promising improvements are empirically selected and tested with, as described below.

Before performing the experiments, some parameters and implementation-related configurations are explored in order to find the optimal settings. Next, three different experiments are performed. The first experiment tests new, promising adaptations to the learning algorithm. The second experiment implements five subskills, including the dynamic task and skill in the domain for the architecture to learn. The third experiment will be expanding the H-DRLN with a dynamic task. The following sections further specify how the tests are performed.

3.1 Initial Parameter Exploration

Before performing the main experiments, it is important to have an effective baseline to compare against. Since this implementation in Malmo differs from the original implementation in BurlapCraft, some elements and parameters might have a different effect on the learning performance of the agent. Preliminary tests confirmed this expectation, where domains such as navigate and pickup were initially unable to reach success rates above 20%. As mentioned above, due to the prohibitive training time required in Minecraft, it is not possible to perform an exhaustive search of the most optimal parameters and settings. The optimal parameters also cannot easily be deduced from logic, since due to the nature of the used methods and algorithms, a small change in settings can have a large effect on the resulting behaviour, as also stated by Mnih et al. in [36]. In order to improve the performance in spite of this limitation, this initial parameter exploration is performed. This exploration is done in an empirical fashion, by searching for 'low hanging fruit', i.e. changes that might have the largest positive effect.

Specifically, the effects of the following parameters are explored: reward normalisation and a zero-based reward distribution are implemented. These should help learning by learning rewards on a smaller scale, and ensures the Q-value is negative for any state-action pair which prevents any positive feedback loops, respectively. Next, the effects of reducing the action set are explored, which could help the agent converge faster since the state space is smaller. Another thing that is looked at is using grayscale versus RGB state data. This further condenses the state space, which either makes it easier to learn, or could make it more difficult to discern differences in states since information is lost. Lastly, the evaluation frequency & length are changed, and the use of a success replay memory is implemented. These changes should allow for more stable, averaged performance monitoring and result in more effective experience replay, respectively.

These parameters are explored for the single room domain, using QR-DQN as a baseline. To measure the effectiveness of these separate adaptations, the success rate of the agent over time is used.

3.2 Experiment I: Algorithmic comparison

In the first experiment, the algorithms DQN, QR-DQN and A2C are compared. This is done for multiple purposes. Firstly, this provides a reference point of the DQN performance compared to the initial H-DRLN study. Secondly, it provides insight into the performance of the newer algorithms compared to each other and compared to simpler games such as the ALE arcade games they are usually tested with. Thirdly, it acts as the deciding factor for which learning algorithm to use to train the individual subskills for the subsequent tests.

The three algorithms are tested in the single room domain. This single room domain consists of an area in which the agent can move is slightly smaller (8*8 tiles), and a gold block is placed in the middle, acting as the goal. The agent can turn 90° left, right, or move forward. The agent succeeds if it reaches (and faces) the goal block, upon which the episode resets. The episode also resets upon reaching the maximum number of actions, set to 30.

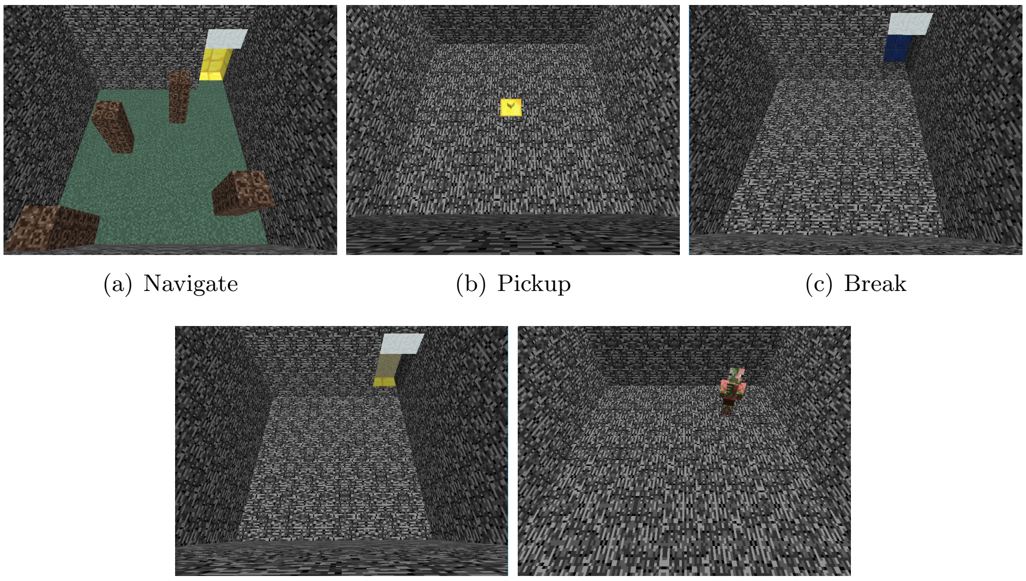
The DQN and QR-DQN are implemented with one agent. A2C makes use of multiple asynchronous agents (called actors) for training, sharing the same model. In the paper where A2C was first introduced, 16 actors on separate threads were employed to train on the ALE games [41]. Due to the more demanding resource requirements of Minecraft as a simulator, it is unfortunately not possible to train with that many actors in this setup. Therefore, only 4 actors will be used to train asynchronously.

Because prior research has found that the more recent QR-DQN and A2C appear to be more stable and efficient in training networks compared to DQN, it is expected that QR-DQN and A2C are more efficient learning algorithms. In order to study the effectiveness of the algorithms, the success rate over time is compared. If QR-DQN or the A2C version of the network is able to converge faster than the DQN version on the same tests, the conclusion can safely be made that the agent learning skills is made more competitive by using these state-of-the-art algorithms.

3.3 Experiment II: (Dynamic) subskills compared

This second experiment is the key contribution from this study and aims to provide an answer to the second research question: the ability of the networks to generalise to more dynamic domains in Minecraft. To do this, after the algorithmic comparison tests have been performed, five separate subskills, including a dynamic subskill, will be trained using the best performing learning algorithm from the previous section.

The four static subskill domains can be seen in Figure 3.1. Each goal is visible in a different way, and different conditions determine the success of the agent. For the navigation skill, the agent simply needs to reach the goal blocks and stand on them. For the pickup skill, the agent needs to reach the goal block similarly to the test domain, and pick up the item. For the break skill, the agent needs to move to the blue blocks, and perform the 'break' action, destroying at least one of the blocks. For the place skill, the agent needs to move to the final goal location (looking similar to the initial navigation goal block) and use the 'place' action. If the agent takes longer than 30 steps, the domain and agent are reset and a new episode starts. These subskills are directly modelled after those in the initial H-DRLN study [5], with as little alterations as possible. This is a crucial part of the research since both: (1) control tests like these contribute to more reliable conclusions and a more cemented research community, and (2) these tests provide the basis for the further experiments and improvements put forward in this study.


(a) Navigate (b) Pickup (c) Break

(d) Place (e) Attack

FIGURE 3.1: The five separate subskills domains in Minecraft which the agent has to learn. Figure (e) is the dynamic task with a monster randomly walking in the space, which is able to hit and kill the player.

For the purpose of this study, dynamic skills are defined as skills that rely on interaction with an NPC (enemy or friendly) or item that can have a variable state, e.g. a friendly NPC can be helpful or indifferent, an enemy NPC can be in different places, and attack or be passive, or an item can depend on other states before it can be used. For this experiment, the following skill and corresponding training domain is chosen: The skill of **avoiding and hitting enemies**. This will be trained in a room of the same size as the other subdomains. In this room, a single monster (a 'ZombiePigMan') is present that tries to attack the agent. If the agent is hit three times, the episode ends and the domain and agent are reset. If the agent manages to hit the ZombiePigMan, the episode ends successfully and the agent recieves a reward. Similarly to the other skills, a small negative reward is given for every action the agent takes, and the episode ends if the agent takes more than 30 actions. The skill is properly learnt when it is consistently able to hit the monster before the end of the episode, translating to a success rate percentage of 80%.

1. The agent has to avoid and hit an enemy that is trying to attack the agent. The enemy can move and hit the agent if it is facing the agent. The episode will fail and start over if the agent loses all its lives, next to running into the limit of the number of steps per episode.

This is a compelling dynamic task, since the skill it conveys can apply to a wide range of situations in more complex and realistic problem domains. An example of how the enemy avoidance is relevant is learning to avoid collisions with other moving entities in the real world. Together with the more static skills previously learnt, learning these skills provides an answer to the first research question, and they are an important step towards more general, capable AI.

The four static subskills and the dynamic subskill are trained separately, using the best performing learning algorithm as decided by the previous section. The episode length for these subskills is 30 steps, the evaluation frequency is initially set to 1k steps, and the evaluation duration is set to 100 steps. The learning rate $\alpha = 0.0025$ and the replay memory size = 100k states. The success rate is the percentage of episodes the agent successfully completed during evaluation.

The training (and testing) of a subskill will be run for over 350k steps. After training, these variations are evaluated and compared to the performance of the framework as reported in [5]. If the results found here match the results reported in that paper within a small error margin, it provides further consolidation as a stable and reliable architecture. If the agent will not have been able to attain a consistent success rate of 80% after training for at most 350k steps, it will be deemed not to have learnt to execute the skill successfully enough.

3.4 Experiment III: training and testing the H-DRLN

After training the subskills individually, if the dynamic skill and at least three of the four static skills that were learnt in the previous experiment each achieve a success performance of at least 80%, a new H-DRLN is trained and tested with both the dynamic and static skills. They are combined in the H-DRLN by use of the Multi-Skilled Distillation. Then they are tested and further trained in a complex domain consisting of three rooms:

1. The first room contains the item that needs to be collected, and has an object break task analogous to that task of the static tests;
2. The second room consists of a navigation task where an enemy needs to be avoided (and hit) to go to the next room;
3. The last room has the exit, where the previously collected item needs to be placed.

Then, the H-DRLN will have all trained DSNs implemented into the architecture and it will be trained and tested using the complex (three-room) domain requiring all subskills to solve it.

If the H-DRLN achieves a succes rate of at least 80% within 750k steps, the architecture will be regarded to be capable of achieving complex goals including learning dynamic tasks.

Chapter 4

Results

4.1 Parameter tests

4.1.1 Initial parameter tests

To explore the possible effects of certain (learning) parameters on the agent, the test domain (single-room) was used where the goal is to perform a specific action when touching the goal block in the room. First, a baseline was trained using QR-DDQN, 200 atoms, an ϵ-decay of 20.000, a replay memory size of 10.000 and an additional success-replay memory. The result of this was an agent slowly performing slightly better and, after around 150K steps, plateauing with a success rate of 53% as can be seen in Table 4.1. This and every subsequent test was ran multiple times and the results are averaged as seen in Figure 4.1.

Next, the effect of reward normalisation was explored by applying a scaling to ensure all rewards lie within the $[-1, 1]$ range. All other hyperparameters were kept the same. Research suggests this can benefit the convergence of the network by learning values on a smaller (error derivation) scale, especially helping transfer learning, by enabling using the same learning rate across multiple domains or games [4, 49]. Results on the 'single room' domain indicate a somewhat steeper learning curve (indicating more efficient learning) reaching a plateau of 72% success rate after around 70K steps.

Lastly, related to the reward normalisation mentioned above, the reward distribution was changed to be zero-based. This means that on success, instead of sending a positive reward signal (of 20), a signal of zero is returned. The negative reward signals (minus one for every action) still remains intact. This ensures the Q-values are strictly negative for any state-action pair, preventing any positive feedback loops that might occur from

Model	**Final success rate**	**Steps until plateau**	**Steps needed for 50% success rate**
Baseline	53%	160.000	180.000
Normalized rewards	71%	100.000	40.000-60.000
Reduced action set	95%	100.000	50.000-60.000

TABLE 4.1: Results of varying hyperparameters on the 'single_room' test domain.

Q-value estimation. The results of this show an increase in performance. In significantly fewer epochs, the success rate peaks higher and more frequently, and the negative fluctuation is less pronounced as well.

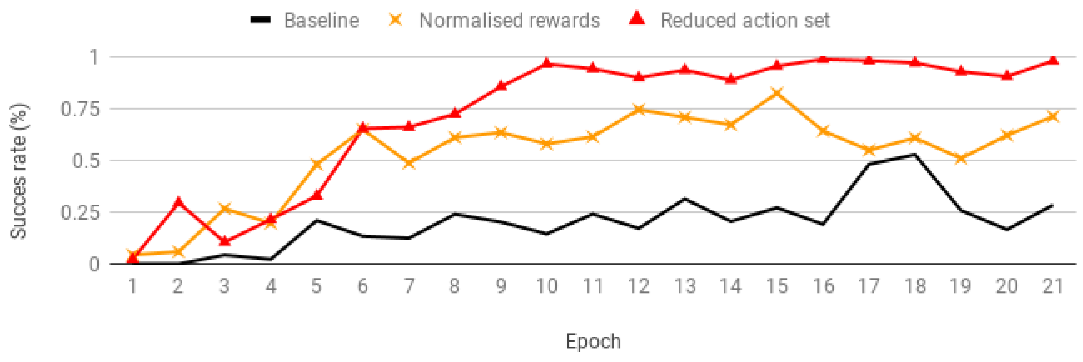


FIGURE 4.1: The success rate of QR-DDQN in the single-room domain for various parameter settings.

Another important factor in the performance of the learning algorithm is the complexity of the explorable state-action space. To test the effect of a reduced action set on the learning performance, the 'move 1' was removed from the agent's available actions. In addition, the rewards are normalised as above. All other hyperparameters were kept the same. Results indicate a similar learning curve as above, but a higher plateau of around 95% success after 100k steps as can be seen in Figure 4.1

4.1.2 Grayscale versus RGB state data

The world state of the agent can be represented with either single-channel (i.e. grayscale) pixel data or full RGB pixel data. Using grayscale results in smaller state sizes, enabling larger replay memories. However, this might cause relevant visual details of the domain that depend on other colours to get lost, impeding learning.

After testing with multiple domains in Malmo, there seems to be a significant difference in the learning effectiveness between grayscale- and RGB-based state data. The results of this for the single room domain can be seen in Figure 4.2. Grayscale state data appears to be much more effective for learning. This might be due to the course resolution of

Minecraft, where visual details are very explicit, even without a higher colour definition. It would be interesting to test the effects in a setting with more realistic graphics (which excludes Minecraft).

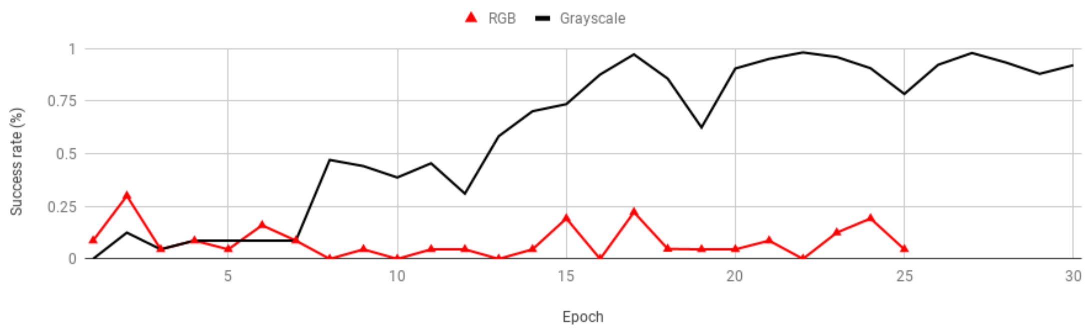


FIGURE 4.2: The success rate of QR-DDQN in the single-room domain with full RGB state data compared to grayscale state data.

4.1.3 Evaluation frequency & length

The evaluation frequency determines for how many steps the agent trains before testing the performance. The evaluation length denotes for how many steps the agent is tested. One such training and evaluation cycle together constitutes one epoch. While not directly influencing the network's ability to learn, it does affect the resulting measured performance. A smaller evaluation length, resulting in fewer episodes played during evaluation, causes more erratic and fluctuating results due to the smaller sample size.

After testing with multiple domains in Malmo, a lower evaluation frequency (10.000 steps instead of 1.000) and longer evaluation length (1.000 steps instead of 100) lead to a more stable learning curve. Consequently, to gain clearer insights in the performance of the agent over many epochs, using a lower evaluation frequency and longer evaluation length is recommended.

4.1.4 Success replay memory

The success replay memory is - next to the experience replay memory used in DQN - an extra memory that stores only the replays of *successful* episodes. When minibatches are sampled from the experience replay memory to train from, it has a chance to sample from the success replay memory instead. As a result, experience replay from successful episodes helps focus on the more important transitions. This can benefit learning especially in a sparse-reward environment such as the ones from this paper.

4.1.5 Conclusions

To conclude the parameters tests, the following settings have been identified to lead to the best performance, and subsequently have been used for the comparison of the learning algorithms and the further learning of the subskills:

1. The rewards are be normalized. This should help convergence of the algorithm.
2. The reward distribution are be zero-based. This should help avoid positive feedback loops.
3. The state data is converted to grayscale, resulting in more efficient memory usage.
4. The evaluation frequency is set to 10.000. The evaluation length is set to 1.000. This leads to a less volatile learning curve.
5. The success replay memory is added to lay more focus on the important transitions when sampling from the replay memory.

4.2 Learning Algorithms Compared: DQN, QR and A2C

The three main different learning algorithms tested with are a 'vanilla' (D)DQN, the more novel QR-(D)DQN, and the parallel learning A2C algorithm. The results can be seen in figure 4.3. In the single room test domain, the DDQN was able to learn the optimal way to the goal, albeit relatively slowly. It took DDQN 10 epochs before it visibly started making progress, and needed 80 epochs to reach a 100% success rate. In a much shorter time frame, QR-DDQN showed a lot more potential, picking up performance after around 5 epochs and reaching a success rate of near-100% after around 10 epochs. Lastly, A2C - which was tested with four parallel agents - proved able to very quickly achieve some level of success, having a success percentage of around 25% in the first epochs already. However in these tests, A2C was not able to achieve a better performance than that initial score, let alone converge to a success rate of 100%. Even after 35 epochs, it only fluctuated in the very low ranges of success, which might be ascribed in part to random chance, and indicates little to no learning taking place. With QR-DDQN clearly performing best, that algorithm was used to further train the separate subskills. Those results are detailed in the next section.

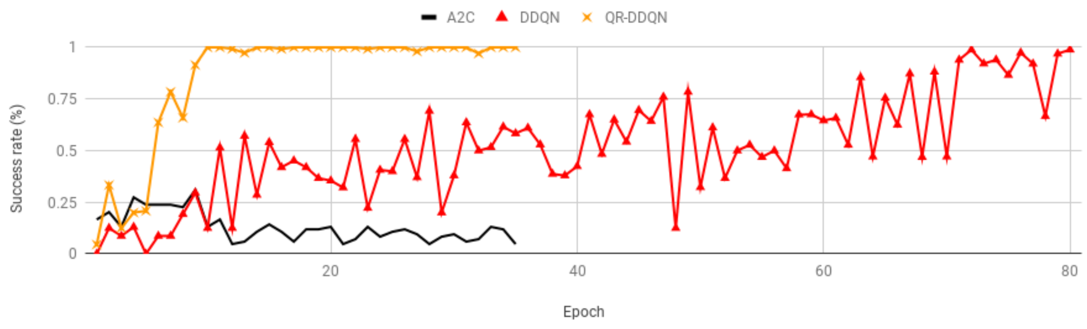


FIGURE 4.3: The success rate of the different learning algorithms (DDQN, QR and A2C) in the single-room domain. They were initially run for 35 epochs, but the DQN was allowed to continue training to explore if and when it would converge, which it did after 80 epochs.

4.3 Domain specific performance in Malmo

The individual subskill domains tests initially show a distinct lack of learning compared to the single-room domain as can be seen from table. Apart from being a more difficult domain, some factors that might contribute to this discrepancy (and for which possible solutions were explored) are the following:

1. The agent has limited guidance from the visuals. Two related elements are identified that influence the network:

   (a) The other element entails the visibility of the goal. Initially even when standing next to the goal in any of the subskills, the goal is hardly visible, being only a gold block in or on the floor. To address this, the subskills pickup and navigation were also tested with more gold blocks placed at the goal's location. The learning progress, compared to the original versions, did not seem to have improved after running the training for 30 epochs, as shown in Figure 4.4 where its effect on the navigation domain can be seen.

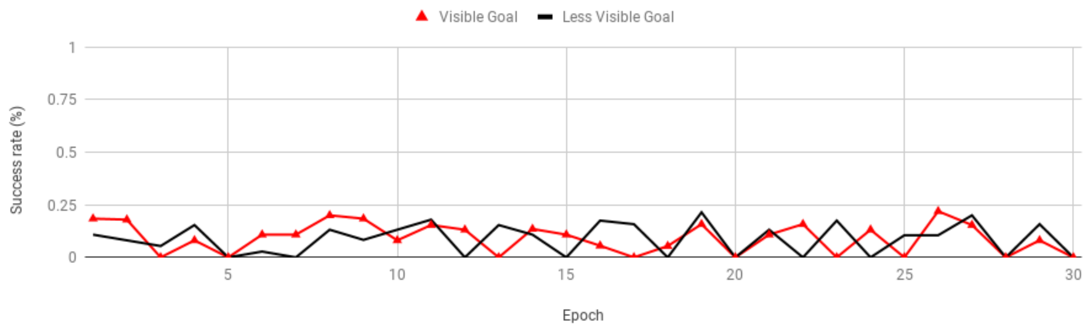


FIGURE 4.4: The success rate of the navigation domain with the goal location varying in visibility.

(b) Secondly, the direction the agent initially faces is fully horizontal. This means blocks close to the agent that are on the floor are hardly visible. This is counter productive, since especially blocks close to the agent are relevant. To counter this, the agent's pitch is tilted downwards by 25°. This way, it can see more details of the lower blocks near the agent, and still sufficiently see blocks at it's own height further off. The results of this after 30 epochs are a slightly more stable and better performing agent, as can be seen from figure 4.5.

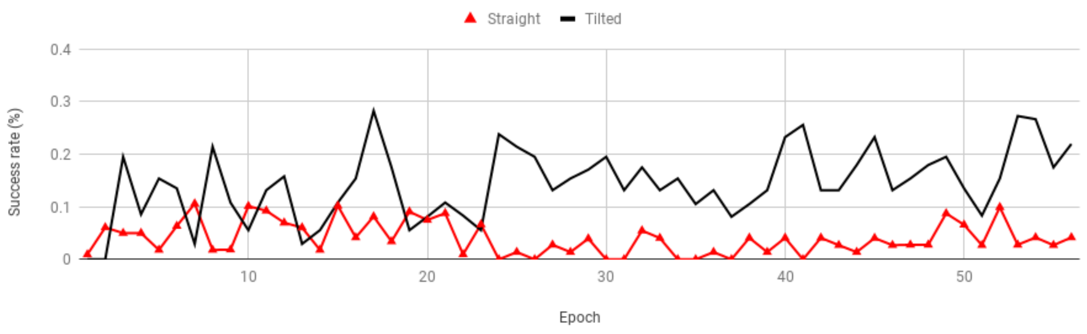


FIGURE 4.5: The success rate of QR-DDQN in the pickup domain with a straight facing compared to a tilted facing.

2. The other contributing factor is specific to the integration with the Malmo platform. Letting the agent itself check if the goal is reached - as used for the pickup, break, and place tasks - provides more control to the agent rather than letting the simulator handle the checks. However, giving the control to the simulator instead seems to provide more stable learning results from empirical analysis. To enable the simulator to take control of the pickup, break, and place tasks, the domains are slightly modified to be more in line with Minecraft's representation of items.

4.4 Subskill performance

The results of the previous section indicate QR-DDQN as performing significantly better than a simple DQN and A2C. Consequently, it was expected that training subskills with QR-DDQN would result in the highest performance. Therefore, the five separate subskills have been trained with QR-DDQN. The results of training both the static subskills (navigate, pickup, place and break) as well as the dynamic subskill (attack) can be seen in Figure 4.6.

The results in the original H-DRLN research for training the static subskills reported attaining success rates of near 100% in around 22 epochs [5]. Based on these results,

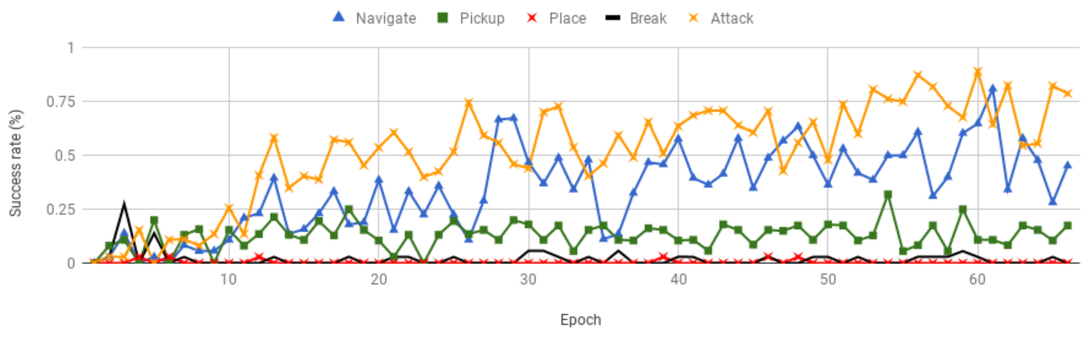


FIGURE 4.6: The success rate of the separate subskills trained with QR-DDQN.

it was expected that the four static subskills trained here would achieve a success rate of above 80% relatively quickly, especially since this study makes use of an improved learning algorithm: QR-DDQN instead of the 'vanilla' DQN (as shown in Section 4.2).

However, in this study, the static subskills did not attain that targeted success rate. The separate subskills have been trained for over 66 epochs (almost twice as long as originally planned), which took around 4.5 days per subskill measured in real time. Some subskills were even trained for over 100 epochs, unfortunately yielding no better results.

The place and break subskills performed the worst of all the trained subskills, never showing any learning progress in the more than 66 epochs of simulation. A poorer result compared to the other subskills was not unexpected since these two skills were also the skills that took one of the largest number of epochs to train in the original H-DRLN study, taking around 22 epochs each [5]. However, the fact that no learning improvement is shown at all over 66 epochs is surprising. Possible causes for this conflicting result can be the differently behaving turn action playing a larger role than initially suspected, or that there were more timesteps per episode needed to initially discover the successful actions for the skill. However, since the rest of the learning parameters are identical for as far as possible, this result was surprising. No other tests have been performed with these subskills. For future research, it is recommended to explore testing these subskills both with longer episode length and in a continuous action space.

The pickup skill quickly reaches a success rate of around 20%, but stagnates after that, showing no further learning over 66 epochs. This result is surprising, since the H-DRLN paper reports this skill reaching a 100% success rate after only two epochs [5], which is very quick. This difference in result may be caused by the different behaviour of the turn action.

Of the static subskills, the navigate skill attained the highest success rate, averaging around 50% near the end. This is surprising, since the goal in that skill domain is often visually obstructed, whereas in the other skills the goal is visible from almost all angles.

This created the expectation that the navigate skill would be more difficult to learn since it hampers the ability to determine the best move towards the goal. This would be in line with the results from the H-DRLN study, where the navigate skill takes relatively long as well to learn correctly, requiring 20 epochs [5]. An explanation for the superior performance to the other subskills might be that the other skills each require a specific action to be performed at the goal area, whereas the navigation skill only required the agent to find its way to the goal.

Multiple tests have been performed with the pickup skill, but this study has not been able to replicate the results from the H-DRLN study. Different setups tested with include other algorithms (regular DQN, QR-DQN (both with and without Double DQN), and A2C), and different domain setups (with and without tilted agent pitch, more visual cues added around goal block). Of those runs, the result shown here was the best resulting consistent performance found. For future work, it is recommended to test with a continuous action space to emulate the differing turn behaviour.

Surprisingly, the dynamic attack subskill performs better than all the other subskills. After 66 epochs, it reached an average success score of near 80%, which was the original goal for the subskills. Since the learning performance seemed not yet to have plateaued, the subskill might be able to attain an even higher success rate, if the simulation had continued to run sufficiently longer. This was not further explored due to time constraints. It was expected to perform in line with the pickup, place, and break subskills which were hampered in their performance by the specific extra action required to attain success. A possible explanation for the attack subskill performing better compared to those is that the monster also moves in addition to the agent. While there are no locations in the state space where the goal is guaranteed to be achievable (in contrast to the static subskills), there now is a chance for all the other locations to be a possible goal-achievable-state. This might have resulted in much more situations where the agent encounters the enemy, accelerating the learning process. Still, this is a promising result for incorporating more dynamic tasks in a Deep Reinforcement Learning application. For future work, it is recommended to train for a higher number of epochs, and explore other dynamic subskills.

4.5 H-DRLN

As discussed in the previous section, after training for over 66 epochs, three of the four static subskills did not reach an average success percentage of 80% or higher, although the dynamic subskill did manage to attain a high enough success percentage. As a result, the H-DRLN as described in Section 3.4 can not properly be trained, since not all subskills will reliantly be able to perform their task. This would result in the H-DRLN agent failing to train which subskill to activate, since the subskills themselves would not result in successes.

However, an exploratory experiment was attempted using a subset of the skills that were able to reach the highest success percentage. Specifically the following subskills were used: the simple single room domain, which is used as a trained skill for the purpose of this experiment, and the dynamic attack skill. These skills reached a success rate of 100% and around 80% respectively. The goal for the agent is to learn to first attack the monster, and then reach goal block from the single room domain.

To perform this adapted experiment, three options for the complex domain layout were considered. With the first option, the domain consists of two connected rooms, one for each of the skills. However, this would require the agent to be able to competently navigate from one room to another. This is one of the skills that was not adequately learnt in the prior experiment. Including this skill would impair the agent's ability to reach the other room and complete the task. Another option was to incorporate both skills in one room. While this does not require other subskill knowledge (avoiding the problem of the first option), it involves non-trivial changes to the skill domains compared to the prior experiments. Specifically, block and monster types other than both skills trained with would be present, resulting in a domain differing from those used for the separate subskills. Because of this change, it effectively also tests the transfer learning ability of the skill at the same time. As an effect, it would be difficult to determine to which of the two changed aspects (i.e. transfer learning versus the skill hierarchy) results of this experiment should be ascribed. The last option was to keep both domains in separate rooms, and teleport the agent from one to the other as soon as the first skill had been achieved. This way, both domains can remain unchanged (avoiding the problems of the first and second option). Also, this method is more in line with both the previous experiments and the original H-DRLN research where the agent is teleported between some rooms of the complex domain as well [5].

To perform this experiment, the last option was chosen to implement and test with, as it was the most promising option for the reasons outlined above. Unfortunately, while the teleportation action used to instantiate the agent in the next room is normally possible within Malmo, it resulted in missing state data for the agent, crashing the simulator. It should be possible to overcome this bug by implementing further checks and a buffer to make sure that state data is available for the agent after teleporting. However, due to the constrained scope of this study, further exploration of the possibilities of the H-DRLN, future research can implement this experiment more in depth.

Chapter 5

Conclusions

5.1 Conclusions

Because of the importance of working towards an Artificial General Intelligence, this study researched the possibilities of learning hierarchical skill structures in Minecraft, a complex open world simulator. The individual skills that would comprise the encompassing H-DRLN skill hierarchy unfortunately were not able to attain adequate performance in the given domains, with the present hyperparameters and algorithms used. As a result, the agent's performance on the complex three-room domain using a hierarchical H-DRLN could not be tested.

This research showcases the effectiveness of DQN, QR-DQN and A2C in a complex simulated world, which is more alike our real world than the traditional ALE simulations. Furthermore, it shows the possibilities of learning dynamic skills. It is a part of moving to more general AI, capable of handling a multitude of complex tasks that can vary wildly in the approach or knowledge required of an agent. This research is helpful for robotics and simulations of agents moving and acting in dynamic environments, interacting with dynamic elements, being able to learn more complex composite tasks.

5.1.1 Algorithmic Comparison

The results from the algorithmic comparison test show a distinct gap in learning performance between the three algorithms. Results show that QR-DDQN clearly performs best compared to the other algorithms. As was expected, QR-DDQN trumps the 'vanilla' DDQN by a large margin, converging to 100% success rate eight times as fast. However, the poor performance of A2C is partly unexpected. The newer algorithm was touted as a clear improvement over the other two [44], but was unable to deliver on its reputation in these tests in Minecraft.

One hypothesis why A2C performs poorly in these tests, is that it's due to the computational load running into the limits of what the set-up tested with can handle. In the tests, it required much more computing power compared to the other algorithms. This was especially visible in terms of RAM required for hosting the Minecraft simulator and the experience replay data. This results in a higher potential for stalls and crashes, impeding the training from a practical perspective. During these tests, the computer was continually at the limit of the available RAM and CPU power, and a hypothesis is that this heavily influenced the learning performance seen in figure 4.3. For example, state- or action data might be lost if there is no more RAM available or if the CPU encounters some race condition from the parallelised agents, impeding the learning process.

Another hypothesis for the poor performance of A2C is that the number of actors was insufficient for the task. The original study presenting A2C as a competitive alternative to established algorithms used 16 concurrent actors. In contrast, in this study, only 4 actors were employed, due to memory constraints as described in the methodology. Using more parallel actors reduces the autocorrelation of the collected (mini)batches by A2C (and results in more exploration). It might be that a higher number of actors can provide the autocorrelation reduction required to get out of learning plateaus for these tests in Minecraft. Consequently, it would be interesting for future work to train with A2C again using more actors - which is possible by using more powerful hardware - to further study its potential compared to the other algorithms in Minecraft.

5.1.2 Subskills performance with QR-DQN

Next to the algorithmic comparison above, one other major contribution of this study is the implementation and analysis of skills having *dynamic* components. This is relevant since dynamic skills are usually more complex to learn compared to static skills. This is especially the case for tabular Q-learning, since the added dimension severely explodes the state space. To investigate the performance of dynamic Deep Reinforcement Learning skills, the original static subskills and a new dynamic subskill have been trained using QR-DDQN since it was the best performing learning algorithm from the previous test.

Surprisingly, the dynamic skill trained was learnt significantly faster and more effectively than the four static skills, reaching a success rate of near 80% while most of the static skills plateaued around 20%. The success curve did not necessarily indicate a plateau, so if the agent had continued to train for more epochs, it might have reached even higher levels of success.

These difference in results between the static and dynamic skill may be explained by the fact that, since the NPC itself was moving around as well, it would more often clearly be in the visual field of the agent, even if it was not yet visible from that location a few timesteps ago. The effect of this might have been that the agent was able to act upon the visual cue more often than with the static skills per episode, where in only one specific area the goal could be achieved.

These positive results indicate that skills relying on dynamic components can very effectively be modelled in this framework using deep reinforcement learning. This has major benefits as it shows an agent can train both in more complex environments and using more complex actions. This shows that Deep Reinforcement Learning is a very promising architecture for learning complicated tasks, and the results would translate better to learning skills in the complex real world.

5.1.3 H-DRLN

While the Hierarchical Deep Reinforcement Learning Network was initially a large focus of this research, the experiments were unable to support further analysis of the H-DRLN architecture. The individual subskills did not all attain sufficient success rates to steer and help train the H-DRLN. Nonetheless, some other very interesting results and contributions have come from this study, as described in the above sections. As such, training with an H-DRLN is left to Future Work, as it is a promising architecture to further broaden the possible complexity of an agent's skills.

5.2 Limitations

Intrinsic to Reinforcement Learning, and still the most limiting factor is the fact that many training steps are needed to reach competent performance, even with the advantages of neural networks. To combat this, parallelisation is key. A prime manner to achieve parallelisation is by using a distributed learning algorithm, where multiple agents train towards the same model, such as employed by A2C or A3C. This would be a relevant direction to explore for future developments.

Another inhibiting point of this research is the use of Minecraft as a simulator. Minecraft boasts an extremely rich and diverse simulated world, enabling very dynamic and complex problem domains. However, due to the software engine used and the fact that

Minecraft wasn't developed with AI research in mind, it is unfortunately a very slow simulator, compared to other platforms (such as ALE). Some of the experiments had to run for up to a full week on a high-end PC before learning stagnated. There are only limited ways to speed up or overclock the Minecraft engine. This is another factor that limits efficient learning for an agent using Minecraft. It would be interesting if Minecraft could be further developed to provide more support for AI research simulations.

Related to the previous point is the Malmo platform itself. Though very promising, for now Malmo still has limited options and control through the available API. The most notable example of this encountered during this study is the inability to accurately reproduce the simulations of Tessler's H-DRLN research [5], compared to which Malmo misses options such as the agent making 30° discrete turns. However, Malmo is still in active development and is open-source, making it very possible to extend the platform with features such as these. Another option is to use continuous movement controls instead of discrete movements.

Lastly, there are the inherent CPU/GPU requirements to using neural networks. To be able to speed up training, the available libraries make use of GPU cores that are optimised for tensor calculations, putting restrictions on which hardware is supported. In the future, CPUs and GPUs will more widely support tensor calculations, but for now (clusters of) high-end (workstation) GPUs are needed to adequately train a neural network.

5.3 Future Work

Next to the *discrete* command set used by this research, Malmo also provides the option of using a *continuous* command set. This provides more human-like input, with for example 'moving forward' resulting in the exact type of continuous movement as a human player would experience, instead of immediately placing the character one discrete unit forward. There are two interesting results of using the continuous command set: (1) it enables more complex interactions in the Minecraft world with monsters and the like. (2) It not only more accurately mimics a human player in *Minecraft*, but more importantly it's also more closely representative of a moving, interacting entity in our continuous *real world*. The continuous command set was not used for this research in the pursuit to initially have as few deviations from the original H-DRLN paper as possible. However for the reasons named above, using continuous movement commands for future work would make an important contribution towards more general AI adequately performing in our real world.

Another thing that would be relevant to further develop is of course training more dynamic skills, and actually training and testing the H-DRLN in a combined three-room domain. Due to the disappointing results of the individual skills, this was not further explored for the combined H-DRLN. An improved learning algorithm or adapted domain interaction might change this.

Next, it would be very interesting to research more complex and realistic situations in Minecraft, such as an open-ended domain world (instead of the enclosed domain space) and less control over the entities that spawn, resulting in the agent encountering real enemies, in varying conditions (of surrounding terrain, daylight, etc). This would provide further insights in the capacity of transfer learning and the generalisation to differing environments.

Lastly, further exploration of the state of the art in reinforcement learning algorithms is necessary. Especially A2C proves very promising, yet challenging to practically execute with such a resource demanding simulation as Minecraft. Future work should test A2C with stronger hardware and more parallel actors. Another new reinforcement learning algorithm with reportedly cutting edge performance is ACKTR, which combines some of the characteristics of A2C with ‘distributed Kronecker factorisation’ - which should improve scalability and sample efficiency - and with ‘trust region optimisation’ - which should result in more consistent improvement [50].

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

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

[3] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 3357–3364. IEEE, 2017.

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

[5] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J Mankowitz, and Shie Mannor. A Deep Hierarchical Approach to Lifelong Learning in Minecraft. In *AAAI*, pages 1553–1561, 2017.

[6] Justin A Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, pages 369–376, 1995.

[7] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[8] Sven Koenig and Reid G Simmons. Complexity analysis of real-time reinforcement learning applied to finding shortest paths in deterministic domains. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1992.

[9] Itamar Arel, Cong Liu, T Urbanik, and AG Kohls. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, 4(2):128–135, 2010.

[10] Vasily Klucharev, Kaisa Hytönen, Mark Rijpkema, Ale Smidts, and Guillén Fernández. Reinforcement learning signal predicts social conformity. *Neuron*, 61 (1):140–151, 2009.

[11] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

[12] Ronald A Howard. *Dynamic programming and Markov processes.* Wiley for The Massachusetts Institute of Technology, 1964.

[13] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719, 1952.

[14] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

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

[16] N Metropolis. Monte carlo method. *From Cardinals to Chaos: Reflection on the Life and Legacy of Stanislaw Ulam*, page 125, 1989.

[17] Leemon C Baird III. Advantage updating. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1993.

[18] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[19] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[20] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[21] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[22] Doina Precup and Richard S Sutton. Multi-time models for temporally abstract planning. In *Advances in neural information processing systems*, pages 1050–1056, 1998.

[23] Daniel J Mankowitz, Timothy A Mann, and Shie Mannor. Time regularized interrupting options. In *Internation Conference on Machine Learning*, 2014.

[24] Timothy Mann and Shie Mannor. Scaling up approximate value iteration with options: Better policies with fewer iterations. In *International Conference on Machine Learning*, pages 127–135, 2014.

[25] Kai Arulkumaran, Nat Dilokthanakul, Murray Shanahan, and Anil Anthony Bharath. Classifying options for deep reinforcement learning. *arXiv preprint arXiv:1604.08153*, 2016.

[26] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 3675–3683, 2016.

[27] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[28] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[29] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in Neural Information Processing Systems*, pages 3486–3494, 2016.

[30] R. Bellman, R.E. Bellman, and Karreman Mathematics Research Collection. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961. URL https://books.google.nl/books?id=POAmAAAAMAAJ.

[31] Cristiano Castelfranchi. The theory of social functions: challenges for computational social science and multi-agent learning. *Cognitive Systems Research*, 2(1): 5–38, 2001.

[32] William Forster Lloyd. *Two Lectures on the Checks to Population: Delivered Before the University of Oxford, in Michaelmas Term 1832*. JH Parker, 1833.

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

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A Brief Survey of Deep Reinforcement Learning. *arXiv preprint arXiv:1708.05866*, 2017.

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

[37] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

[38] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[39] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[41] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[42] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

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

[44] Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. OpenAI Baselines: ACKTR & A2C, 2017. URL `https://blog.openai.com/baselines-acktr-a2c/`.

[45] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016.

[46] Chen Tessler. Malmo reinforcement learning environment. `https://github.com/tesslerc/malmo_rl/`, 2018.

[47] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[48] Krishna Aluru, Stefanie Tellex, John Oberlin, and James MacGlashan. Minecraft as an experimental world for ai in robotics. In *AAAI Fall Symposium*, 2015.

[49] Hado P van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295, 2016.

[50] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5285–5294, 2017.

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