Using Reinforcement Learning and Task Decomposition for Learning to Play Doom

Ivan Kropotov

Helsinki June 5, 2020

UNIVERSITY OF HELSINKI
Master's Programme in Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Study Programme in Computer Science

Tekijä — Författare — Author

Ivan Kropotov

Työn nimi — Arbetets titel — Title

Using Reinforcement Learning and Task Decomposition for Learning to Play Doom

Ohjaajat — Handledare — Supervisors

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	June 5, 2020	49 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Reinforcement learning (RL) is a basic machine learning method, which has recently gained in popularity. As the field matures, RL methods are being applied on progressively more complex problems. This leads to need to design increasingly more complicated models, which are difficult to train and apply in practice.

This thesis explores one potential way of solving the problem with large and slow RL models, which is using a modular approach to build the models. The idea behind this approach is to decompose the main task into smaller subtasks and have separate modules each of which concentrates on solving a single subtask. In more detail, the proposed agent will be built using the Q-decomposition algorithm, which provides a simple and robust algorithm for building modular RL agents. The problem we use as an example of usefulness of the modular approach is a simplified version of the video game Doom and we design a RL agent that learns to play it.

The empirical results indicate that the proposed model is able to learn to play the simplified version of Doom on a reasonable level, but not perfectly. Additionally, we show that the proposed model might suffer from usage of too simple models for solving the subtasks. Nevertheless, taken as a whole the results and the experience of designing the agent show that the modular approach for RL is a promising way forward and warrants further exploration.

ACM Computing Classification System (CCS):
Computing methodologies → Machine learning → Learning paradigms → Reinforcement learning
Computing methodologies → Machine learning → Machine learning algorithms
Computing methodologies → Artificial intelligence → Distributed artificial intelligence → Cooperation and coordination

Avainsanat — Nyckelord — Keywords

reinforcement learning, modular reinforcement learning, multi-objective reinforcement learning

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms study track

Contents

1 Introduction 1

2 Reinforcement Learning 3

2.1 The Reinforcement Learning Setting . . . 3
2.2 Rewards and Policies . . . 4
2.3 Exploration-exploitation Dilemma . . . 6
2.4 RL Algorithms . . . 7
2.4.1 Q-learning . . . 7
2.4.2 Sarsa . . . 8
2.5 Function Approximation in RL . . . 9
2.5.1 Linear Function Approximation . . . 10
2.5.2 Other Function Approximation Methods . . . 11

3 Modular Reinforcement Learning 11

3.1 Motivation . . . 12
3.2 Definitions . . . 12
3.3 Representative Approaches . . . 13
3.4 Q-decomposition . . . 14

4 Video Games as Research Platforms for AI 15

4.1 Motivation . . . 15
4.2 Overview of Video Game Research Platforms . . . 16
4.3 ViZDoom . . . 17
4.4 Doom . . . 18
4.5 Previous Doom Agents . . . 18

5 Agent Architecture 19

5.1 State Representation . . . 20
5.1.1 Object Grid Representation . . . 21
5.1.2 Distance Grid Representation . . . 22
5.2 Object-Based Modules . . . 22
5.3 Area Exploration Module . . . 25

6 Experimental Setup 26

6.1 Task Description 27
6.2 Training and Testing Environments 27
6.3 Training Procedure 29

7 Results 30
7.1 Performance on the Training Environments 31
7.2 Performance on the Validation Environment 36
7.3 Visual Inspection of Trained Agent Performance 38

8 Analysis of Q-decomposition Performance 39
8.1 Performance Comparison of Different Training Methods 39
8.2 Analysis of the Performance Drop Caused by Finetuning 40
8.3 Shortcomings of the Proposed Agent 42
8.4 Practical Considerations When Using Q-decomposition 44

9 Conclusion and Future Work 45

1 Introduction

Reinforcement learning (RL) is a machine learning approach, were the goal is to learn how to solve a problem by interacting with the environment and trying to maximize the received numerical feedback [33]. This approach has had many successes in recent years, with applications ranging anywhere from robotic manipulation [9] to playing video games [13, 19, 25]. These accomplishments were typically achieved using single monolithic learning systems, that were trained end-to-end on a single task. However, once the focus moves onto more and more complex problems, traditional learning systems may not be enough to cope with them in large part due to the curse of dimensionality.

The curse of dimensionality in RL means that the state space of the problem grows increasingly fast as the task gets more complicated. To illustrate this, we can think of a binary vector representation of a state, with two, four and sixteen features. Now, with two features we would end up with $2^2 = 4$ states, with four features we would have $2^4 = 16$ states, with sixteen features $2^{16} = 65536$ states and so on. Naturally, when the state space is large representing it and searching in it becomes increasingly difficult or even infeasible. Consequently, large state spaces necessitate the usage of function approximation, which may lead to further optimization difficulties as the interesting functions may be very difficult to learn due to their shapes. Dividing a large problem into smaller, more manageable, parts is a standard approach in computer science, which may be a potential solution to building learning systems for increasingly complex tasks.

The idea to divide RL problems into smaller pieces is not new and has been previously done in various ways. These methods can be roughly divided into two classes. The first one is hierarchical reinforcement learning (HRL), where the task is divided into hierarchy of temporally abstracted (macro-)actions, i.e. we allow actions to take multiple time steps and consist of smaller actions [34]. The other way is sometimes referred to as modular reinforcement learning (MRL) where the idea is to decompose the original task into subtasks, use a separate model learn to solve each of the subtasks and then to combine the learned behaviors in some meaningful way [12, 26, 28, 31]. In this setting separately doesn't necessarily mean that the subtasks are optimized individually, and often learning happens in parallel.

Reinforcement learning techniques have traditionally been evaluated on simple toy problems to show their correctness and performance. However, to truly appreciate the benefits of more modular approaches and to evaluate the applicability and limits of these methods in real life tasks, the problems to be solved should be complex enough. In other words, while toy problems are extremely useful in research, these results may correlate poorly with applicability and performance on real world problems, due to a toy problem's contrived nature and possible research bias in the experiment setting. To battle the shortcomings of the simplest of toy problems, video games have been proposed, and used, as a test platform for reinforcement learning and AI research in general [4, 10, 17, 19, 23, 25]. Usage of video games in research is motivated by them having many desirable properties for a test platform:

1) Video games are easy to run on computers, which makes experimenting cheaper and easier, 2) One can find a video game of any desired difficulty and complexity, 3) Video games are designed to be played and enjoyed by humans and primarily not as an experiment, which means that the task is complex enough to be difficult for humans and additionally might help in alleviating the research bias.

Some of the popular testing platforms based on video games are the Arcade Learning Environment (ALE) [4], where the task is to play various classic Atari arcade games, and ViZDoom [17], where the agent must learn different tasks in an environment with dynamics from a video game called Doom. Among these and other current video game-based research platforms, the ViZDoom platform provides a very complex problem, which makes it an interesting case for RL research.

In this thesis, we will explore one way to transition towards a more modular approach for RL models, where each module would have its own state representation, potentially coming from a separate ML model, and concentrate on solving one problem well. To do this a modular RL agent for playing a simplified version of the original Doom will be presented. Doom was chosen as the main task, because in our opinion the problem must be complex and varying enough to truly see the benefits of MRL, and Doom happens to provide such task in a convenient way via the ViZDoom library [17]. More specifically, a model based on the Q-decomposition algorithm [28, 32] is designed and implemented. This algorithm was chosen due to its simplicity as well as theoretical guarantees of its convergence and optimality [28]. The proposed model comprises of modules that solve specific subtasks, e.g. combat, navigation, item gathering, etc., and a way for choosing which action to perform. Each module will have their own reward signal and state abstraction function to reduce the state space and make the subtasks easier to learn.

Additionally, we also explore a few different ways such modular RL agent could be trained so that the process would be more efficient and produce better results. These proposed training methods mainly concentrate on pretraining the modules separately and then finetune them all together, which should reduce the total training time as the individual module pretraining can be done in parallel.

The thesis is structured in the following way. We begin by introducing reinforcement learning together with the related concepts and issues, including two basic algorithms, in the Section 2. Section 3 introduces modular reinforcement learning and the motivation behind it, provides a short overview of related approaches and introduces the central algorithm of this thesis called Q-decomposition. Section 4 tells about using video games for AI research, explains the video game Doom and motivates its usage in this work. The architecture of the proposed Doom playing agent, i.e. modules, rewards functions and state representations, is presented in detail in the Section 5. Section 6 describes the final task that the agent must learn and provides a detailed description of the how the agent will be trained. The detailed results of the performance of the proposed agent are presented in the Section 7. Section 8 analyses the gathered results and provides some explanations to the questions raised by them. Finally, Section 9 contains the final thoughts and outlines potential directions for the future work.

2 Reinforcement Learning

In this section reinforcement learning and related topics are introduced. The section starts by introducing and defining reinforcement learning in general after which some of the related concepts are introduced in more detail. After this two relevant RL algorithms, Q-learning [38] and Sarsa [27], are presented and in the end of the section we explain function approximation methods in RL at a level needed for understanding the rest of the thesis.

Reinforcement learning (RL) is a machine learning paradigm that focuses on learning from interaction. The problem we are trying to solve is how to learn action strategies, or policies, to reach the goal while being in a potentially unknown environment. The difficulty is that a priori we do not know the outcome of the actions and whether they are good or bad from the perspective of reaching the goal. However, we do get some feedback that tells us something about the outcome of the chosen action after it has been executed.

The inspiration for reinforcement learning comes from the field of psychology, where the idea of trial-and-error learning is from. Some of the methods are also inspired by concepts taken from neuroscience, and conversely some of the reinforcement learning algorithms have found uses in that field. Additionally, reinforcement learning is closely related to dynamic programming and optimal control. For a thorough introduction to reinforcement learning, including the history of the field, see e.g. Sutton and Barto [33].

2.1 The Reinforcement Learning Setting

Reinforcement learning setting can be thought of as having two interacting entities: the agent and the environment. The agent is the one that actively acts in the environment by modifying its state by taking actions. The environment on the other hand contains everything that the agent cannot change arbitrarily and reacts to the agent's actions by changing its state and providing feedback.

To make the idea clearer, we can think of a now popular self-driving cars as an example. Our task is then to design an agent that can drive a car from one place to another. In this example the agent would be the computer that can turn the wheel and press the pedals. The environment is everything else, including the car, since the agent cannot for example set the speed of the car in an arbitrary way, but the wanted pace must be reached by a sequence of appropriate gas or brake pedal presses and waiting for the physical state of the car to change.

Now, the basic interaction loop between the agent and the environment is very simple. In discrete time case, at each timestep t the agent gets an observation of the current state of the environment s_t and chooses some action a_t. After the action has been executed, the environment reacts to it and changes its state. Finally the environment gives the agent a numerical feedback, called reward, r_{t+1} and the new state s_{t+1} that resulted from the agent's action.

As for most problems in science, it is very useful to have a mathematical model of the problem. For reinforcement learning problems the model that is almost universally used is called (finite) Markov decision process (MDP) [11]. Formally MDP is defined as a tuple (S, A, δ, R), where

- S is a finite set of states.
- A is a finite set of actions.
- $\delta : S \times A \times S \to [0, 1]$ is a (stochastic) transition function, which tells the probability of moving from state s to state s' after choosing an action a.
- $R : S \times A \times S \to \mathbb{R}$ is a reward function which associates a scalar number to a state transition.

The main assumption behind the MDP model is that the state transition has a Markov property, i.e. that the next state is fully defined by the current state and the chosen action. The Markov assumption is quite a strong one and unfortunately doesn't hold on most of the interesting problems. Nevertheless, MDPs are have been shown to be a very useful tool for both theoretical analysis and building practical algorithms.

The usefulness of the, MDP model is emphasized by its abstractness and flexibility for modeling RL problems with different assumptions. The way we have defined MDP above is for the most basic discrete and fully observable cases. However, the formalism can be extended to cover continuous state and action spaces, partially observable environments (POMDP) [2] and multi-objective problems (MOMDP) [26], as we will see and use later in section 3.

2.2 Rewards and Policies

The overall setting of reinforcement learning is quite simple and intuitive, but how exactly do we make the agent do what we want? In RL there is an underlying assumption, called the reward hypothesis. This hypothesis states *"That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)"* [33].

Using the reward hypothesis, all goals for RL agents are defined by the given reward signal. The reward must define which outcomes are favorable and which should be avoided from a perspective of reaching the goal. For example, continuing with the self-driving car problem, we could give a positive reward to the agent when the car reaches its destination and give a large negative reward if the car crashes into something. By trying to maximize the reward given this way, the agent will learn how to get to the set destination without crashing, which is exactly the goal we set for the agent.

So, the goal of a reinforcement learning task is defined as maximization of the cumulative reward. To formalize this idea, we introduce the (discounted) return G_t, which is the sum of rewards starting from timestep t

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k}. \tag{1}$$

In this definition T is the final timestep, i.e. the timestep when the terminal state is entered, and an episode ends. This of course only makes sense for episodic tasks, or continuous tasks where we only want to optimize for a finite horizon. However, this leads to poorly defined return if the interaction doesn't end, since if $T = \infty$ the return could become unbounded. To remove this possibility we have the discount factor $\gamma \in [0, 1]$. It controls the current value of future rewards with $\gamma = 0$ meaning that only the next reward is taken into account and increasing discount rate results in agent taking future actions more into consideration. By disallowing setting both $T = \infty$ and $\gamma = 1$ we have a well-defined notion of return. In practice the return in both episodic and continuous task is written with $T = \infty$ and adding an absorbing state which is entered from terminal state from which any action results to returning to the terminal state with a reward of zero.

The purpose of learning in RL is to find a strategy of choosing the actions so that the cumulative reward is maximized. In RL such strategy is called a policy π and it is a mapping from state to actions, or to action probabilities. Formally a policy is a function $\pi : S \times A \rightarrow [0, 1]$, which for a given state returns the probability of an action, which in deterministic case is zero for all but one of the actions.

For a given policy π we can then define value functions that describe the expected consequences of being in a state via the expected reward. Two of the most important value functions are state-value functions V^π and action-value functions Q^π.

Informally the state-value function expresses how good a given state is from a perspective of solving the task. Formally, a state-value function V^π for a policy π gives the expected return when starting in a state s and following the policy π afterwards:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t | s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s \right]. \tag{2}$$

The action-value function, also known as state-action-value function and Q-function, is similar to state-value function, but it tells us the desirability of an action in a given state instead of desirability of a state. Formally, the action-value function for a state-action pair (s, a) is the expected return after choosing an action a in the state s and following policy π thereafter:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t | s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a\right]. \tag{3}$$

As we have stated before, solving a reinforcement learning problem means finding an optimal policy π^* that maximizes the expected discounted cumulative reward $\mathbb{E}\left[G_t\right]$. Since value functions define a partial ordering over policies with a policy π being better π' when $V_{\pi}(s) \geq V_{\pi'}(s) \quad \forall s \in S$, we can define the optimal state-action function as

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{4}$$

and equivalently optimal action-value function as

$$Q^*(s) = \max_{\pi} Q^{\pi}(s,a). \tag{5}$$

Now, we can find an optimal policy by first finding an optimal value function and the acting greedily with respect to it, i.e. we always choose the action that has the largest action-value in each state. This is called action-value method because the resulting policy is defined by the learned value functions. The relevant algorithms for finding optimal policies, which fall under the action-value methods, will be introduced in section 2.4. Naturally, there are other approaches to RL, like policy gradient, but these are out of scope of this thesis.

2.3 Exploration-exploitation Dilemma

At the heart of RL is the idea of learning from interactions, but how to interact in a way to get enough information is not straightforward. On one hand we would like to explore a lot to see as many states as possible so that we wouldn't miss any promising ones. On the other hand, pure exploration will lead to a policy that often chooses non-optimal actions, which is not desirable from a point of view of achieving a good performance in the task. This leads to a problem called exploration-exploitation dilemma, which means that we must somehow balance exploring new states and exploiting already learned knowledge to find a good solution.

Obviously, we want the learned policy to be such that exploits its knowledge as much as possible while still looking for new, better ways of achieving its goal. So, we would like to have our policy to stay explorative, but such policy cannot be optimal in general. However, we are always learning about a policy that we are following, which leads us to a problem where we would like to act according to an explorative policy but learn about the optimal one.

In RL there are two ways of approaching this problem. The first one is to use a single policy and bake exploration into it, so that the policy is close to optimal but still explores a bit. This method is called on-policy learning, because in it the policy that is learned about is the same that is used to choose the actions.

The other approach is to use one, exploratory, policy for choosing the actions, called behavior policy, and another policy that we are actually trying to learn, called target policy. This is called off-policy learning, since we are learning about one policy using the data collected from following another policy. Off-policy learning might seem superior to on-policy learning, and indeed it is in some respects like in ability of using any policy for data generation. However, off-policy methods also typically have larger variance, slower convergence and in some cases even divergence (see Deadly Triad [33]). Additionally, for some problems on-policy methods are just better suited, one example of which will be seen further in the thesis (Section 3.2).

Exploration in both off and on-policy methods is often achieved using an ϵ-greedy strategy. The idea of the method is quite simple, given a small $\epsilon \in [0, 1]$ the agent chooses the greedy action with probability of $1 - \epsilon$ and a random action with probability of ϵ. This rule has the effect of the agent trying some non-optimal actions even though it acts greedily most of the time. Often, the ϵ term is reduced as the learning goes on, so that the agent would concentrate more on exploitation rather than exploration as the training progresses.

To illustrate the differences between on-policy and off-policy learning we can once again think of the car driving example. If we would use on-policy learning, we would have to gather the data and behave using the same policy, which sometimes may make a non-optimal exploratory action. However, the agent is aware of this and will learn to compensate for such behavior. On the other hand, if we use off-policy learning, we could use any driving data available to us, e.g. from other human drivers, and improve our policy using that data.

2.4 RL Algorithms

During the last few decades of RL research many different algorithms have been developed. In this section two algorithms, Sarsa and Q-learning, are introduced that are important for the methods developed in the subsequent chapters. Both algorithms are based on action-values and are model-free, which means that they do not use a model of the transition dynamics of the environment. Algorithms in this chapters are also assumed to be tabular, in other words the action-value function can be represented as a table of $|S|$ rows and $|A|$ columns, i.e. each row represents the estimate of action-value function for one state and each action.

2.4.1 Q-learning

We will start with perhaps one of the most widely known RL algorithms called Q-learning [38]. This is an off-policy model-free algorithm that approximates the optimal action-value function Q^*. The algorithm is quite simple: at each timestep the agent chooses an action a, transitions from the current state s to the next state

Algorithm 1: Q-learning

```
1 Initialize $Q(s,a)$ for all pairs $(s,a)$ and initialize the first state $s$
2 repeat
3     Choose $a$ using $\epsilon$-greedy rule
4     Observe $r$, $s'$
5     $Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
6     $s \leftarrow s'$
7 until $s$ is terminal
```

s', receives reward r and updates the estimate of the action-value function of the current state s according to the Q-learning update rule:

$$Q(s,a) = Q(s,a) + \alpha\left[r_t + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{6}$$

where γ is the discount rate and $\alpha \in]0,1]$ is the learning rate (sometimes called step size). Pseudocode of the algorithm is shown as Algorithm 1.

In the equation (6) the quantity $r_t + \gamma \max_{a'} Q(s',a') - Q(s,a)$ is called temporal difference (TD) error. It measures the error between the current value function estimate $Q(s,a)$ and the more recent and accurate one $r_t + \gamma \max_{a'} Q(s',a')$, which is sometimes called TD target. Intuitively the algorithm iteratively updates the estimate of the action-value function by pushing the estimate towards the TD target with the intensity of the update controlled by the learning rate.

The exploration policy in Q-learning is typically just an ϵ-greedy strategy with respect to the current estimate of action-values. The target policy is then such, where $\epsilon = 0$.

Q-learning has a nice property of provably converging to an optimal policy in tabular case [39]. The policy found by the algorithm converges to an optimal policy in the limit as long as learning rate satisfies Robbins-Monro conditions and all the state-action pairs are visited infinitely often in the limit, i.e. $Q_t(s,a) \to Q^*(s,a)$ as $t \to \infty$ if α decays appropriately.

2.4.2 Sarsa

Another widely known RL algorithm is Sarsa [27]. Like Q-learning, Sarsa is a action-value based and model-free algorithm, but it is on-policy instead of off-policy. The origin of the algorithm's name is related to its update rule, since for the update it uses current state s, current action a, the reward gained from transition r, next state s' as well as the next action a', whose symbols combined form the name Sarsa. The Sarsa update rule is quite similar to Q-learning with a difference that it also uses the next action for the update, which is chosen by the current policy thus making the algorithm on-policy:

Algorithm 2: Sarsa

```
1 Initialize $Q(s,a)$ for all pairs $(s,a)$ and initialize the first state $s$
2 Choose $a$ based on $s$ using $\epsilon$-greedy rule
3 repeat
4     Observe $r$, $s'$
5     Choose $a'$ based on $s'$ using $\epsilon$-greedy rule
6     $Q(s,a) = Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
7     $s \leftarrow s'$, $a \leftarrow a'$
8 until $s$ is terminal
```

$$Q(s,a) = Q(s,a) + \alpha\left[r_t + \gamma Q(s',a') - Q(s,a)\right]. \tag{7}$$

So, instead of taking the best possible, i.e. greedy, action from the next state, this algorithm uses the estimate that results from the action that will actually be taken as determined by the behavior policy. The pseudocode for Sarsa is shown as Algorithm 2.

Sarsa also provably converges to an optimal action-value function in tabular case [27]. The conditions for convergence are similar to Q-learning, i.e. state-action pairs are visited infinitely often and learning rate decays fast enough, but with additional condition that the policy must become greedy in the limit, i.e. the exploration rate must diminish as the learning goes on.

2.5 Function Approximation in RL

The tabular methods introduced in the previous chapter have many nice properties: they are simple and easy to prove theoretical results on. Additionally, they work very well on simple problems and provide the most precise value functions for the problems they can be used on. However, it is easy to see that using tabular methods on larger and more complicated problems is not feasible, since the state and/or action spaces will grow fast.

The obvious consequence of the growth of the state space is that at some point we will run out of memory to hold the value-functions in. Another, perhaps even more important aspect, is that as the state space grows the agent must visit an increasing number of states in order to learn good value function. Ultimately visiting all of the states enough times, or even visiting all of them at least once, becomes computationally very costly and eventually intractable. To battle this problem, we need a way to generalize the knowledge gathered from the visited states to new, similar states.

This generalization can be achieved using function approximation, which has been successfully used in RL. The goal of function approximation is to find a function that resembles the target function as much as possible. The function used for approximation is usually chosen from some family of parameterized functions, so the goal is often to find the right parameters.

In RL function approximation is typically used to find a parameterized approximation to a value-function, i.e. to find some parameters θ s.t. $f_\theta(s) \approx V(s)$ for all states s. Because the number of parameters is typically much smaller than the number of states, changes to the parameters often affect values of multiple states, that are hopefully similar, through which the generalization effect is achieved.

2.5.1 Linear Function Approximation

The simplest function approximation scheme, that will be extensively used in this thesis, is linear function approximation (LFA). In this method, we must define a feature vector for each state $\mathbf{x}(s) \in \mathbb{R}^n$, that describes the state, and a weight vector $\mathbf{w} \in \mathbb{R}^n$, which the agent must learn. Now the value function, e.g. state-value function, is approximated by the inner product between the weight and feature vectors:

$$\hat{V}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s), \tag{8}$$

where the parameter $\mathbf{w}$ signifies that the state-value function approximation is parameterized by the weight vector $\mathbf{w}$.

The weights of the linear function are often optimized using stochastic gradient descent (SGD), because it is simple and effective method. In SGD we iteratively update the weights in the direction of the steepest decent, which is the negative gradient of some loss function. The stochastic part of SGD come from the fact that the gradient is calculated based on a random sample from the data distribution, which is often useful as it is impossible to compute the gradient using all the data points.

In particular, when using linear function approximation the gradient w.r.t. the weights of the approximate value function is really simple: $\nabla \hat{V}(s, \mathbf{w}) = \mathbf{x}(s)$. As an illustrative example, here is the Sarsa update rule from Equation (7) with linear approximation:

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \alpha \left[r_t + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right] \nabla \hat{Q}(s, a, \mathbf{w}) \\ &= \mathbf{w} + \alpha \left[r_t + \gamma \mathbf{w}^T \mathbf{x}(s', a') - \mathbf{w}^T \mathbf{x}(s, a) \right] \mathbf{x}(s, a). \end{aligned} \tag{9}$$

Linear function approximation is a very compelling method due to its simplicity, but at the same time that simplicity makes it quite limited. The reason for that is that in this method the feature vectors are assumed to be independent, which means that any interaction between the features is not accounted for. Interactions can be modeled with careful function construction, e.g. using polynomial basis where the interaction between features x_i and x_j can be modeled by adding $x_i x_j$ as a feature to the feature vector. There are of course many other ways to construct features for linear approximation ranging from coarse coding to Fourier basis and combinations of those (see e.g. [8, 18]). However, these methods are far from automatic, and the feature selection and construction must be done carefully, which means that it usually takes a lot of effort to engineer the right state representation.

2.5.2 Other Function Approximation Methods

Another popular function approximation technique is artificial neural network (ANN) which is a non-linear function approximator and thus more powerful than linear function approximator introduced previously. The topic of ANNs and their usage in RL is very broad one and goes beyond the scope of this thesis. However, it is still useful to introduce them as they are very popular at the moment and their computational complexity serves as a motivation to using simpler methods.

An early and notable success in using ANNs with RL is due to Tesauro who used multilayer ANNs to teach an agent how to play a game of backgammon with minimal prior knowledge already in the 1990's [35, 36]. More recently the success stories of using deep neural networks with RL, e.g. learning to play classic Atari games from raw visual information [25] and learning to play Go [29, 30], have reinvigorated the research in the field. The widespread usage of deep ANNs has been motivated not only by their non-linearity but also by the ability of deep neural networks to perform a kind of automatic feature extraction from the raw input, which potentially makes it easier to design the features.

However, on the downside ANNs are computationally very demanding and typically require extremely large amount of data and time to learn, which is not very surprising as these are very flexible models. This leads to the problem with sample inefficiency, i.e. deep RL methods require enormous amounts of interactions to learn, e.g. millions in Atari games [25], which makes them difficult to use in many practical applications.

Other types of function approximation have been used as well, including nonparametric methods, but unfortunately all these methods are out of scope of this thesis. A short review of different function approximation methods used for RL can be found for example in in [33].

3 Modular Reinforcement Learning

In this section modular reinforcement learning is introduced and its connection to multi-objective reinforcement learning is illustrated. Some representative approaches are described together with an introduction of the Q-decomposition algorithm that is in the main component of the Doom playing agent that is proposed this thesis.

3.1 Motivation

As mentioned before, one of the main problems with RL is the curse of dimensionality, which arises from having either a large state space, action space or both. One way to try to reduce the problem with large state spaces is to somehow reduce their sizes. Obviously, one could just remove some features from the states, but this would lead to diminished performance especially if we assume that the states already mostly contain only crucial information needed for solving the task. However, if we assume that the task can be decomposed into a number of subtasks, so that each subtask doesn't require all of the available information. Then we can use only a subset of features for solving each subtask, since we can ignore the features a subtask doesn't depend on.

For example, say that we want to build an AI whose goal is to drive a car to some destination as fast as possible while following the traffic rules. This task could be decomposed into two tasks *DriveFast* and *FollowRules*. Now, *DriveFast* agent wouldn't need to pay attention to the speed limits as its only goal is to drive as fast as possible, while *FollowRules* probably doesn't need all the information provided by the handling system of the car. In this example, we could then reduce the state space of both subtasks by removing the features that are unneeded for solving them. This way we get both the benefit of smaller state space and possibly an easier function to optimize for when using function approximation, as we have potentially less irrelevant noise terms.

3.2 Definitions

The idea of decomposition of the full task into smaller pieces is the motivation behind an approach sometimes called modular reinforcement leaning (MRL). More specifically, in MRL each subtask is learned by a separate module and the action to be carried out is selected using some arbitration rule based on the outputs of the modules in a way that should provide an optimal policy for the full task. The term MRL itself is not very well defined in the literature and is sometimes used interchangeably with multi-objective reinforcement learning (MORL) that is a field which tackles problems that consist from multiple, possibly conflicting objectives [22, 26]. However, MORL is, arguably, a more general problem that includes the case of computing a set of policies that contain an optimal policy for any prioritization of the subtasks. Consequently, this field contains MRL as a special case: it is similar to the known weights scenario in MORL [26]. In this work we will use MRL to mean specifically the approaches where the idea is to reduce computational complexity of a problem via utilization of task decomposition and having a separate agent for each subtask.

Since MRL is so closely related to MORL, we will use the notation and some tools from that field to give a more precise description of the MRL problem. The setting of MRL, and MORL, is that we have decomposed our task into multiple subtasks that each have their own reward function and possibly state representation. To model this setting, we can generalize the standard MDP model from the single objective case into multi-objective case. This model is called multi-objective MDP (MOMDP) and differs from a standard MDP by having a vector valued reward function $\boldsymbol{R} : S \times A \times S \to \mathbb{R}^n$, where each component is a reward associated with a subtask [26]. Since the rewards and by extension the value functions are now vectorized, it is not clear anymore what is the best policy. This is because now we can have a situation where for a single state one policy may have larger value for one subtask while having smaller value for another subtask than another policy. This means that we cannot choose between such policies unless we know the priorities of the subtasks. To choose between tasks we can transform the vectorized value functions into scalar ones by using a scalarization function f which encodes the priorities among the subtasks in some way. The choice of a suitable scalarization function is a nontrivial question but is often chosen to be a linear combination of the vectorized value function with some weights. And indeed, all of the MRL methods presented in this work assume that the scalarization function is linear.

3.3 Representative Approaches

There have been many proposed RL algorithms that fall under MRL approach. Before describing the Q-decomposition algorithm in detail this section will introduce some other relevant MRL methods and motivate the choice of Q-decomposition.

One of the simplest approaches to MRL is to use a separate switch that chooses, using some technique, e.g. Q-learning [21, 31], the module that will be allowed to choose the next action . In this method the switch, called arbitrator, is a RL agent itself that has its own state space and reward function and its actions consist of choosing among the modules. This approach has very few restrictions, e.g. we can choose the learning algorithm(s) more freely (e.g. Sarsa or Q-learning) [31], and the design of reward functions is easier as we do not have to assume the additive property of reward, i.e. linear scalarization function. However, this method introduces more design choices to consider when designing the arbitrator, which include designing an appropriate state representation for it. In general, to choose the optimal module for each state, the arbitrator should have all the information it needs for such a decision, which may include all the available state features. Since our goal is to make the tasks easier to solve via decomposition, creating a very complex arbitrator is counterproductive. Due to this issue, it is very difficult to apply this approach to an actual problem, since for it to make sense we should either have an extremely large action space, choosing from which would be delegated to modules, or some task where most of the state features are not needed for module selection. This downside is the main reason why it was decided not to use this kind of an approach in this work.

Humphrys [12] has explored different arbitration techniques one of which, sometimes called modular Q-learning, is using Q-learning to learn action values for each module and choose the action that maximizes the sum the modules' action values (similar approach also studied by Karlsson [16]). However this approach is problematic, as discussed in [32] and [28], and can lead to a suboptimal policy. This problem, called illusion of control, follows from the properties of the Q-learning update rule 6, due to which each module assumes that it can choose all of the future actions. In reality, however, the arbitrator is the one choosing the actions which leads to potentially suboptimal policy.

To counter the illusion of control Russel & Zimdars [28] and Sprague & Ballard [32] propose using Sarsa instead of Q-learning. Sarsa being an on-policy algorithm doesn't suffer from this problem, since each of the modules use the arbitrator's policy as the behavior policy. Moreover, Q-decomposition converges to the optimal policy, at least in the tabular case [28]. The whole Q-decomposition algorithm is described in detail in section 3.4.

The ideas behind Q-decomposition were later combined with artificial neural networks and general value functions by van Seijen et. al. [37]. This approach is quite interesting, but was not pursued in this work, since the idea of this thesis was to use the as simple models as possible and show that complex methods such as ANNs are not necessary. Another notable example is the W-learning by Humphrys et.al [12]. This method while interesting was not explored here due to time limitations.

3.4 Q-decomposition

Q-decomposition is an MRL algorithm whose main idea is to simplify the agent construction and improve task learning by decomposing the full task into subtasks which are learned by separate modules. The name Q-decomposition is from [28] but it is also called modular Sarsa(0), e.g. in [32]. The underlying assumption behind the algorithm is that we have a linear scalarization function, i.e. the reward function of the full task can be expressed as a sum of the subtasks' rewards. As mentioned before, this setting is equivalent to MORL with known weights.

More formally the problem is defined similarly to a standard MDP formalization, but with addition of modules. We begin by decomposing the main task into n subtasks, which gives us as many modules. This gives us a vector valued reward function $\boldsymbol{R} : S \times A \times S \to \mathbb{R}^n$. Now, for each module $j \in 1, \ldots, n$ we define the modules' action value function as

$$Q_j(s,a) = \mathbb{E}\left(R|s,a,\pi\right) \tag{10}$$

The action selection at each state s is performed based on the action-values returned by the modules. The selected action a is the one that has the largest combined action-value i.e.

$$a = \arg\max_{a \in \mathbb{A}} \sum_{j=1}^{n} Q_j(s,a) \tag{11}$$

The state-action values for each modules are learned with a standard Sarsa update

Algorithm 3: Q-decomposition

```
1  Initialize $Q_j(s,a)$ for all pairs $(s,a)$ and modules $j$ and initialize the first state $s$
2  Choose $a$ based on $s$ using $\epsilon$-greedy rule
3  repeat
4      Observe $r$, $s'$
5      for $a' \in A$ do
6          $Q(s',a') = \sum_{j=1}^{n} Q_j(s',a')$
7      end
8      Choose $a'$ based on $s'$ using $\epsilon$-greedy rule based on $Q(s',a')$
9      for module $j \in 1,\dots,n$ do
10         $Q_j(s,a) = Q_j(s,a) + \alpha\left[r + \gamma Q_j(s',a') - Q_j(s,a)\right]$
11     end
12     $s \leftarrow s'$, $a \leftarrow a'$
13 until $s$ is terminal
```

rule (7) adapted for multiple modules,

$$Q_j(s,a) = (1-\alpha)Q_j(s,a) + \alpha\left(r(s,a,s') + \gamma Q_j(s',a')\right) \tag{12}$$

where the next action s' is the actual action chosen by the arbitrator using (11). The usage of an on-policy update rule like Sarsa, which uses the actual reward chosen by the arbitrator for action-value update, removes the illusion of control present in Q-learning. In tabular case Q-decomposition has the same optimality and convergence guarantees as standard Sarsa, i.e. it will converge to the optimal policy with the standard conditions [28]. The pseudocode for the Q-decomposition is shown as Algorithm 3.

4 Video Games as Research Platforms for AI

This section explains motivation behind usage of video games in AI research as well as introduces some notable examples. Here we will also introduce the video game Doom, which is used as the main task in this thesis as well as the library called ViZDoom [17], which enables the interaction between the agent and the game. In the end a few previous notable AI agents that were proposed for playing Doom are presented.

4.1 Motivation

Development of AI algorithms requires having good benchmark problems that are easy to experiment on while being complex enough to show that the algorithm that is being studied is capable of solving difficult tasks. Traditionally simple toy problems were used in research that were quite limited and often had to be designed for testing a particular algorithm. While these kinds special purpose experimental environments are undoubtedly useful in research, they are often not complex or general enough, or are difficult to implement or build. Special purpose benchmarks may also introduce research bias, where a researcher can (subconsciously) design an experiment that will influence the results in a certain way.

Video games can be considered as a type of toy problems, since they are typically heavily abstracted representations of the real world with objectives that often relate to some real-world tasks. However, as they are designed to be played and enjoyed by humans, games contain many complex and challenging tasks, solving which take quite a lot of effort even from human players. Additionally, video games are designed to be run on computers and typically have standard implementations, which makes using them fairly straightforward. And, perhaps most importantly, video games provide quite constrained environments, at least compared to the real world. This enables the researchers to concentrate on solving specific problems, instead of forcing to cope with everything that an agent could face in a real world. Furthermore, video games often provide different settings, which change the nature of the tasks, e.g. difficulty setting, and this can be used for starting with simpler cases and gradually continue towards the full-fledged problems.

4.2 Overview of Video Game Research Platforms

The properties given in the previous section, make video games seem as promising platforms for research and development of AI approaches. And indeed video games have been used in AI research since at least the early 1980's [23] and are continued to be used more and more to this day with promising results [13, 19, 20, 25, 41]. To facilitate the development of AI algorithms a number of different games and platforms based on them have been proposed. Some of the more notable ones are presented here, among other things to serve as a comparison to ViZDoom which is introduced in section 4.3.

Currently, perhaps the most widely known video game based research platform is the Arcade Learning Environment (ALE), which provides a suite of classic Atari 2600 games [4]. This platform has a large number of games, that are very simple and abstract and can be solved even by quite simple methods. However, it has become the de facto benchmark for RL research, that many seminal works have used, including the famous Deep Q-learning paper [25].

Deepmind lab [3] is a more complex research platform based on a game called Quake III, though heavily modified. This platform provides a visually rich 3d environment that can be used for solving various tasks including navigation and simple puzzles.

Obstacle Tower is a recently proposed benchmark for AI systems, which consists of navigation tasks and light puzzle elements that are performed in a rich 3D environment [14]. This platform provides good API for interacting and a way of randomizing details in the levels, which helps in avoiding overfitting and it has tasks that require memorization. However, this is not strictly a video game platform but rather "game-inspired", since it is not based on any existing video game.

Lastly, roguelikes, which is a genre of very challenging games with high degree of randomness, have been used for testing AI algorithms since at least the 1980's [23]. Recently there have been numerous papers using or advocating for usage of roguelikes as a testbed for AI and reinforcement learning in particular [1, 6, 7, 15].

4.3 ViZDoom

Most of the recent and famous approaches that used video games as benchmarks used very simplistic video games like those for the Atari 2600. These games while being quite difficult to humans, are still extremely simple and do not really resemble any real-life tasks. Other platforms provide more complex environments, but either modify the base game very heavily or are not video games in the strict sense at all. To provide a solution to this problem a research platform based on the classic video game Doom was proposed, which is called ViZDoom [17].

The main goal of ViZDoom was to provide a platform that is more complex than some of the popular environments, while being closer to the real world by being a 3D environment and enabling easy experimentation with visual learning. It provides a fast implementation that is computationally efficient while being extremely flexible in terms of the design of experimental setting. While the main emphasis is on visual learning, i.e. learning from raw visual information, ViZDoom provides easy access to other game information besides the frame buffer, e.g. object position and bounding boxes.

The environment provided by ViZDoom has the player, or the agent, that sees the environment from a first-person view and the task is to navigate and/or survive in a complex 3D maze-like environment while battling enemies. The possible task settings range from simple navigation and collection objectives to battling other computer-controlled enemies and solving light puzzles, or any combination of these. The environments along with the tasks can either be easily built by the experimenter with the readily available tools or premade environments can be used, with the latter being helpful with upholding reproducibility of the experiments.

Most importantly, ViZDoom has an excellent API that makes interacting with the game easy and has built in tools for input preprocessing, e.g. depth information. Additionally, ViZDoom can provide different kinds of auxiliary information, e.g. objects seen on the screen, which are helpful in constructing state representations. The different tasks, scenarios and levels can be easily constructed with provided tools and they can even be randomly generated by using additional software. The large number of different environments makes it easy to separate them to training and testing sets in order to assess the algorithm's generalization capability and to reduce the effect of overfitting. The flexibility of the platform also means that it is possible to convert the more standard visual RL setting into a setting that utilizes object information as was required by the methods used in this thesis.

4.4 Doom

Since ViZDoom is based on the game Doom, its dynamics, appearance and typical goals are identical or very similar to the original. So, before introducing the particular tasks and environments that the final agents are trained on, which will be done in chapter 6, it is useful to introduce the game here.

Figure 1 shows an example screen from the game that has many of the gameplay elements. The player has few resources, which are health, armor and ammunition. Health is the most important quantity whose depletion means game over. It can be gained by picking up medikit, the small white box in Figure 1, and will be drained by the enemy attacks and various elemental hazards, e.g. lava floors. Armor is a helpful bonus that can be picked up during the gameplay which reduces the amount damage, i.e. reduction in health, that is inflicted on the player. Ammo is self-explanatory as it describes the number of projectiles for any given weapon that the player carries. In addition to these, the player also has a collection of weapons, that can be used on enemies, and keys, that are used for opening locked doors in the level. Doom also has some special items, but these are not important for our purposes.

The task of the player is to navigate to the exit of the level, whose location is typically unknown in the beginning, without dying. In any given level, there are also enemies that are of different types with different behaviors, whose task is to stop the player by attacking him. Besides the enemies and maze-like environments, there are also doors that require finding keys from the level before they open. Combining all these features, makes Doom quite challenging even for a human player and remains a largely unsolved problem by any AI approach [41]. Therefore in this thesis we will concentrate on solving a simplified version of a full Doom game that is closely related to the limited deathmatch task in [41] which will be introduced in detail in chapter 6.

4.5 Previous Doom Agents

There have been quite a few Doom playing agents proposed during the last few years, in part because of the ViZDoom competitions that were held [41]. Most of the proposed ViZDoom agents were made using some different kinds of neural network based architectures, reinforcement learning and typically had some hardcoded rules. Some notable examples are presented in this section, but for a more comprehensive survey see [41].

Lample and Chaplot made a reinforcement learning agent that used a modular architecture [19] that had two neural networks for different phases of the game that were trained using DQN [25] using different reward functions. The first network was a simple ANN, used for map exploration. The other was a more complicated recurrent neural network, which allowed for rudimentary memory, that was used for battling enemies. The network to use for action selection was chosen via a

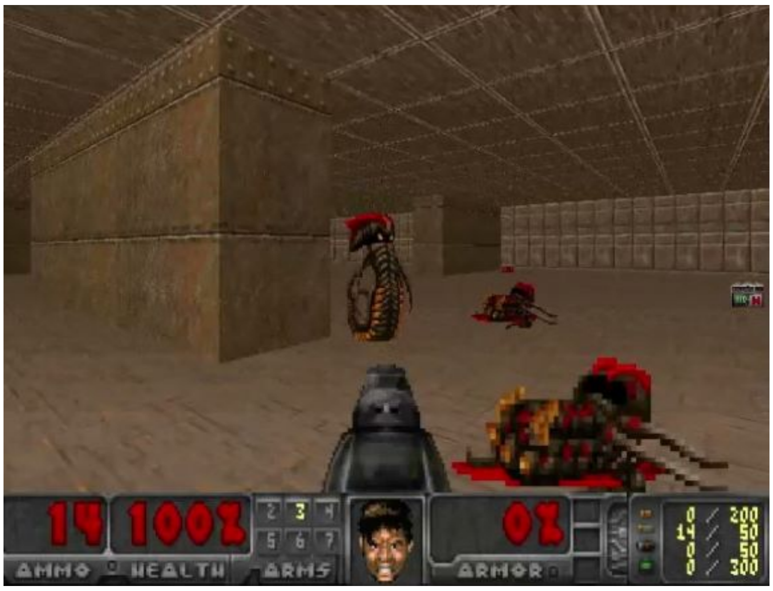

Figure 1: A typical screenshot from a Doom level. At the bottom of the screen are shown the attributes of the player. In the middle an enemy is seen. On the right there is a medikit and in the far middle back there is an ammo pack.

simple rule: the battle network was used when an enemy was seen on the screen and exploration network was chosen otherwise. Ground truth object information was used in the training phase and together with some hardcoded rules, e.g. the agent was programmed to be always crouching.

Wu et.al developed a deep RL agent [40], that was trained using the A3C algorithm [24]. The "trick" behind their approach was in using curriculum learning, i.e. training the agent on a sequence of increasingly difficult training environments. This is a similar idea to the "task rotation" training evaluated in this thesis, In "task rotation" the agent is trained on different maps that are designed to train specific skills. However, the difference here is that they used the same task with different difficulty levels instead of different tasks altogether as in the method proposed here.

5 Agent Architecture

This section provides a detailed description of the proposed architecture of the agent. The agent is built using the Q-decomposition method, which means that it is inherently modular in its design. Each module of the agent is designed for learning a single task, which was chosen to be as simple as possible to make it learnable with an extremely simple method, which in this case is linear function approximation. The task decomposition used in this work produced the following subtasks: ammo gathering, health gathering, armor gathering, shooting and area exploration. Agent has one module for each of the tasks and the construction of the modules will be described in this section. An overview of the agent architecture is shown in the

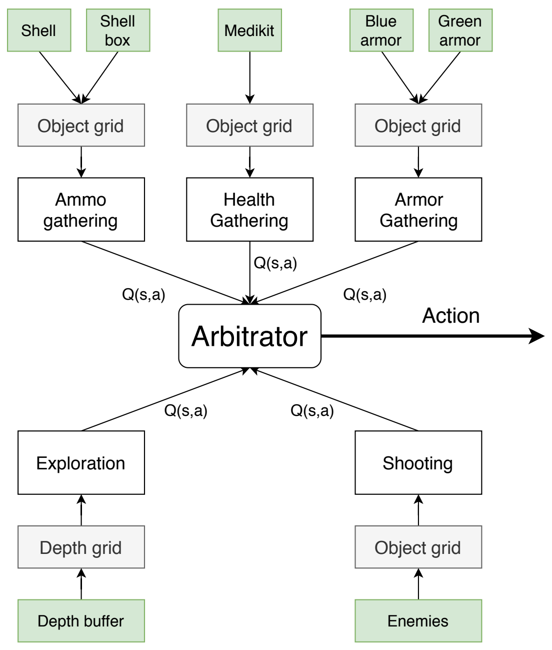

Figure 2: A diagram representing the architecture of the proposed agent. The white boxes represent the modules, grey boxes the state representation they use and the green boxes show the main information included in the state. The arbitrator represents the act of choosing an action in Q-decomposition, i.e. the choice of the action that has the largest summed action value across the modules.

Figure 2.

5.1 State Representation

The engineering of the state representation is one of the most important, and difficult, tasks of RL and choosing a particular function approximation method can restrict the choice of suitable representations. For linear function approximation, the one considered here, there are many ways of constructing the state ranging from tabular representation in the degenerate case to using complex basis functions like Fourier basis [18].

The representation used for the modules is one of the most basic ones, being quite close to tabular representation in simplicity, called fixed sparse representation (FSR) [8]. In this representation method each state is mapped to a binary vector where each component is representing a presence of a feature. Each feature in a given state can be active independently of the others, which reduces the potential state space when compared to the tabular case, where each combination of features must be represented as a separate state. More formally FSR is a mapping $\mathbf{x} : S \to \{0,1\}^k$ where $\mathbf{x}(s) = (x_1(s), \ldots, x_k(s))^T$ and each x_i indicates the presence of a feature i in the state s. FSR is quite limited state representation method and can be a bad choice in general [8]. However, it proved to be quite a good match for the item-position based state representation we chose for most of the modules.

5.1.1 Object Grid Representation

One natural representation of a state is through objects, and their attributes, that are present at that moment. Luckily it is easy to locate the items relevant to each task through the ViZDoom API and get their attributes. Therefore, the positions of the objects are used as the basis of state representation for most of the modules. Because the item positions, i.e. coordinates, are continuous values, discretization is used, which leads to a grid-like state representation. This kind of representation will be referred to as object grid representation.

In detail, the state is represented as a grid, that divides the space in front of the agent into small sectors. Polar coordinates are used instead of the standard Cartesian coordinates, because they lend themselves better for the chosen representation. With polar coordinates an object's position with respect to the origin, in this case the agent, is defined by the distance and the angle from it, denoted by ρ and ϕ respectively. Thus, the coordinates of an object are written as (ρ, ϕ), where $\rho > 0$ and $\phi \in]-\pi, \pi]$.

Since both of the coordinate values are continuous, they must be discretized in order to utilize them as features in the FSR based state representation. This is done by dividing each dimension into a finite number of intervals, or "bins", and using the interval as the approximate coordinate. For example, if we have an interval for angles $[j, k]$ then all angles in that interval are mapped to the same approximate coordinate i using the discretization function, i.e. $\forall \phi \in [j, k], \phi \to i$.

When the discretization is done for both dimensions, the resulting bins produce a grid, which is visualized in Figure 3. Each such grid cell, or rather its occupancy, is a feature in the corresponding FSR. This way, in state s if there is an object in a cell i then $x_i(s) = 1$ and zero otherwise. Since the state vector is binary, it can only represent the fact that there is an object in a given cell but not the number of objects present in that cell. This is a clear limitation of the chosen method, but it can be sidestepped by choosing the grid cells to be small enough, which is seemingly enough for the tasks considered here as will be seen in chapter 7.

The object grid representation has a few hyperparameters that can be chosen for each module separately depending either on prior knowledge or experimentation. For range ρ we have the maximum range max_r, which represents the maximum distance the agent can detect objects at. We can define the resolution of the grid in "ρ"-dimension by choosing the number of bins n, that divides the interval $[0, \mathrm{max\_r}]$ into intervals of equal length. Since the values of the angle ϕ are bounded, we only have the number of bins k as a hyperparameter, which divides the range of possible angles into k bins of equal size. Additionally, we can add an extra constant feature to the state, that would represent a bias. This feature allows to get a nonzero value from the linear function approximation, when all of the actual features are zero and can be useful in some cases.

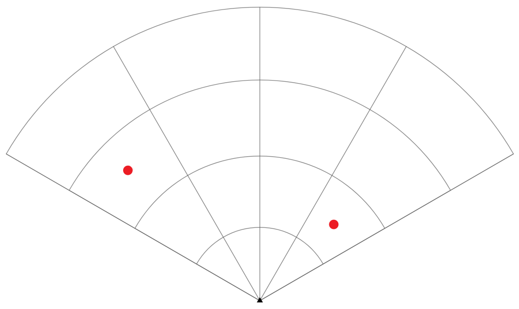

Figure 3: Schematic representation of a object based state representation. Red dots indicate the presence of an object of interest in that sector. This means that the binary features corresponding to the occupied sectors will have a value of one.

5.1.2 Distance Grid Representation

The state representation for the exploration task needs a different approach and so is different from object grid representation, though it still utilizes FSR. Since the goal is covering as much ground as possible, it is very important not to get stuck in the geometry of the level, i.e. walls and other obstacles. This means that the agent must see the walls and other obstacles around it so that it could learn to avoid them. The representation presented here uses information of the distance to walls and objects and will be referred to as distance grid representation.

Since we are constrained to using only linear function approximation, using the raw video information is infeasible. Luckily, ViZDoom provides a depth buffer, which shows the distance to an object or a wall from the player. The frame buffer is provided in a way that is visualized as an image in Figure 4. Since the depth buffer is very large, using it in its entirety with linear approximation is still unpractical, but we can use small parts of it.

To build the representation, we choose a small number of values along one row in the depth buffer. These points can be thought as "laser rays" that measure a distance at that point. These "rays" return distance to a point on screen which, being a continuous value, is discretized into bins, whose number is a hyperparameter. So, the final state representation consists of $n \times k$ binary features, where n is the number of "rays" and k is the number of bins used for discretization of the distance values".

5.2 Object-Based Modules

This section explains the design of the modules, whose representation only depends on object positions and use the object grid representation. These modules are health, armor and ammo gathering modules as well as the shooting module.

Picking up different items is one of the basic skills needed for playing Doom, since most of the interactions deplete the finite resources of the player, e.g. ammo when shooting, that can be replenished only by gathering more from the level. There are

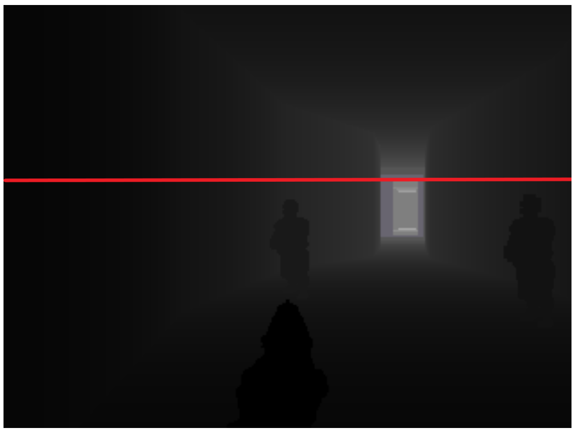

Figure 4: A visual representation of a depth buffer. The lighter the pixel, the further it is from the player. The red line represents the row on which the "rays" are located.

quite a few items in the full game of Doom, ranging from new weapons to temporary invulnerability, but in the limited setting considered in this work only three main item types are kept: ammunition, health and armor. Ammunition and armor both have two different variants that give different amount of the corresponding resource, while health can be gained only from a single kind of item.

Collecting each of the item types can be considered a separate task, and indeed this is how it is used here. However, the state representation for each of the item types is very similar, so the module structure of these tasks is the same. In fact, the resulting optimal policy, as represented by value functions, should be very similar between the tasks, albeit with different numerical values due to differences in the reward scale. This means that, in theory, one could try and use the same action values for all of the three tasks if the values are scaled properly, though it may be that the different reward scales could result in significantly different value functions. In any case, in this work it was decided to learn all the weights separately to investigate if the Q-decomposition algorithm can handle a larger number of modules.

Perhaps the most important skill needed in Doom is shooting, thus it has its own module. To be able to shoot at the target one only needs to see the target and know its location from the shooters point of view. Therefore, the shooting module only pays attention to the enemies seen on screen. Since the enemies are objects, the shooting module uses an object grid representation as well.

The state representation for each of the object-based modules is based on the object grid representation, however there are small differences in the final state representation. The differences between the modules are mainly in the item types they see and how they handle the resource cap, i.e. the maximal amount of a resource a player can have, and not having some resources. The object-based modules must keep track of the resources, since for example if the health capacity is full the agent cannot pickup more health so going towards it is pointless. More importantly if the agent doesn't know that it has full health, then not being able to pick a health item up is completely arbitrary from its point of view. This implies that without the resource information the state representation does not have all the needed information for solving a task. So this information is also included in the modules via rules when to ignore the items, though these rules should be seen as additional state information provided to the agent.

The summary of how each of the object-based modules sees its world is provided in the following list:

- Ammo gathering module
  - This module sees the two item types that give ammunition: Shell (4 ammo units) and Shell Box (20 ammo units).
  - If the current ammo count is at least 30, then the Shell Box items are ignored (maximum ammo count is 50).
  - If the current ammo count is at least 47, then all items are ignored.
- Health gathering module
  - This module see the only health related item: Medikit (25 health units).
  - If the health is full (100 health units) then all items are ignored.
- Armor gathering module
  - This module sees the two items that give armor: Green Armor (100% armor) and Blue Armor (200% armor).
  - When the armor is at least at 100%, then the Green Armor is ignored.
  - When the armor is at least at 200%, then all items are ignored.
- Shooting module
  - This module only sees enemies.
  - If the agent has no ammunition, all enemies are ignored.

To complete the description of the object-based modules we need to define the reward signal. Because of the task decomposition the subtasks became very simple and the reward signals for the object-based modules are quite straightforward.

For the item gathering tasks, the agent is rewarded with a fixed positive reward each time it collects the wanted item, e.g. medikit for health gathering module, etc. Each of the items has a bit different reward due to importance of the item, e.g. health is more important than ammo because losing all health means game over.

The reward for the shooting task is more complex, with the agent receiving a positive reward for killing an enemy and a smaller negative reward for shooting, which discourages the agent from wasting the finite ammo. The positive reward of shooting module is chosen to be the highest, since destroying demons is the most important thing to do in order to achieve good performance in the task.

All of the object-based modules also receive a so called "living" penalty, which is a small negative reward given at each timestep regardless of the chosen action. This kind of reward is quite often used in reinforcement learning, and here it was observed to improve the performance of the modules. The reward signals of all of the modules are shown in table 1.

Subtask	Reward
Ammo gathering	Picking up "Shell" or "Shell Box" +10 Living reward -0.001
Health gathering	Picking up "Medikit" +10 Living reward -0.001
Armor gathering	Picking up "Green Armor" or "Blue Armor" +10 Living reward -0.001
Shooting	Destroying an enemy +100 Shooting (when ammo is used) -10 Living reward -0.001
Exploration	Change in distance to the current center divided by 100 Living reward -0.001

Table 1: The reward signals for each subtask.

5.3 Area Exploration Module

Efficient exploration of the map is extremely important for a good performance, since finding new items and enemies is impossible by staying at the same place. The importance of this task has also been noticed in previous work where exploration was for example encouraged via reward shaping [19, 40] or induced using hardcoded rules [41]. In the proposed agent, exploration is considered a separate subtask with a goal of getting as far as possible from the previous position. The reward signal of the exploration task is similar in its idea to displacement reward used in [19, 40], where a reward proportional to the distance traveled since the last step is given to the agent to encourage movement.

To calculate the reward, we define a displacement circle with a center point (x_0, y_0), which is set to be the current position of the agent, and a radius r, which is a hyperparameter. The reward then is the change in the distance from the center point since the last step scaled by some constant, here 100^{-1} is used to avoid overflow issues. This means that if the agent moves toward the center point, the reward will be negative, and if it moves away from it, reward will be positive. When the agent gets out of the displacement circle, i.e. its distance from the current center point is larger than r, the center point is updated to the current position of the agent.

The reward defined this way encourages the agent to move away from the position it was recently at, while allowing it to revisit old areas after some time. The choice of the radius of the displacement circle is an important one, since by setting it too small, the agent can fail to move far away, while too large radius can result in getting stuck, e.g. by being unable to leave dead ends. After some experimentation the radius was chosen to be equal to 210 in all experiments.

Finally, the area exploration module is provided a small shaping reward to further discourage bumping into objects. When any of the "rays" shows that the distance is under some threshold, the agent is given a small fixed penalty. Both the threshold and the penalty are hyperparameters that are chosen by experimentation. As with the other modules, a small "living" penalty is added to the module's reward, which in this case discourages it from staying still. Summary of the reward function is shown in Table 1.

6 Experimental Setup

In this section the experimental setup is explained. The goal of the experiments is to provide answers to the two of the research questions. The first question is that can the proposed agent learn the task and how good can it get, and the second is how the training method affects the learning process and the final performance. To assess these questions the agent will be trained on the ViZDoom environments that were specifically made for the experiments. To evaluate the performance, the score gathered by the agent during the training and testing phases will be collected, since the score directly measures the level of performance of the agent. For more qualitative assessment of the agent's performance videos of trained agents performing the main task will be recorded.

The expected results follow the hypothesis presented in the beginning. The agent should be able to learn to play the game quite convincingly, but not perfectly, with performance close to a simple game AI or a novice human player, since each of the subtasks are quite easy and should be possible to learn using the chosen methods. However, due to simplicity of the proposed models, it is not expected that the agent will be able to come up with any complex strategies that require memorization or reasoning thus not being able to get to the expert human level. Due to model's simplicity it is also expected that it will show some non-human-like behavior, e.g. forgetting items and enemies that were in its view just a second ago. As for training methods, the simpler methods should result in slower learning and worse performance. This means that for example the methods utilizing task rotation, i.e. the agent is trained on different (sub)tasks in an alternating fashion, should outperform those that don't and agents using warm start, i.e. whose pretrained weights are finetuned, should outperform those that don't.

The structure of the section is following. First the task that is intended for agent to learn will be described in detail. After this the used environments, i.e. Doom maps, will be introduced. In the end a detailed explanation of the training procedures and hyperparameters will be given.

6.1 Task Description

The task to be learned is related to the limited deathmatch setting from [41] but with a few crucial differences. Originally it was intended to use the standard limited deathmatch, but due to technical issues and time limitations the current setting was devised.

The main goal of the agent is to destroy as many enemies as possible before its health reduces to zero, which ends the episode. The enemies are all of the same type, namely "imps", that have the basic Doom behavior provided by the original game AI. The agent interacts with the environment via six actions that are SHOOT, TURN LEFT, TURN RIGHT, MOVE RIGHT, MOVE LEFT and MOVE FORWARD. Choosing an action is comparable to pressing a button corresponding to that action on a virtual keyboard.

The agent is limited to having only one weapon, which is a shotgun. The shotgun was chosen instead of a rocket launcher that was used in the limited deathmatch setting in [41] due to technical issues with ZDoom. In addition to shooting enemies, the agent can collect ammo, health and armor bonuses, which are scattered across the map in predefined locations. The enemies spawn continuously according to a schedule, but the items do not respawn. This leads to an eventual end of the episode as the agent will inevitably be overrun by enemies after exhausting all of the ammunition present in the map.

The performance metric for the main task is the score gathered during the episode, which is calculated as the number of the destroyed enemies multiplied by 100. Since the goal is to destroy as many enemies as possible, higher score implies better performance.

6.2 Training and Testing Environments

The environments used during the experiments, also referred to as maps and scenarios, can be divided into two classes with the first being single task setting and the other full task setting. The single task environments are, as their name suggests, designed for learning a single (sub)task, while the full task environments require mastering all the subtasks to get a good score. All the environments, regardless of their class, share the same reward signals, available actions and object information, as well as environment dynamics. The scores used to evaluate performance in each of the scenarios are closely related to the reward signal of the agent as defined in section 5. First, we will go through the single task environments and then full task environments.

The environments designed to teach different item gathering tasks are similar in structure. They represent a large rectangular hall with a fixed number of items of a single type scattered around the room. The episode ends when either all the items are gathered, or the timeout is reached. The score of the episode is the number of gathered items multiplied by 10.

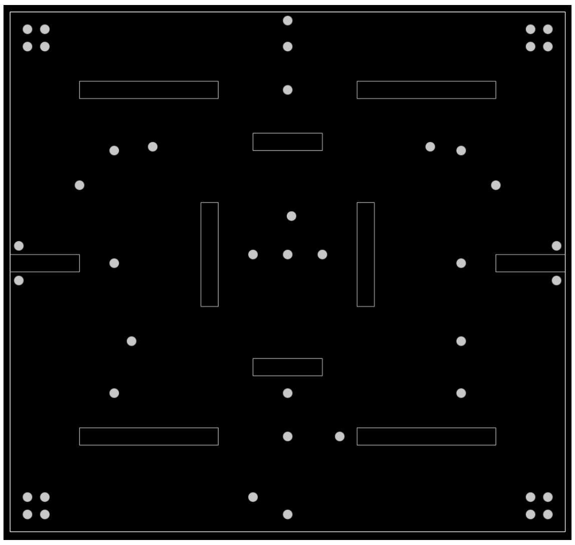

Figure 5: Full task training map. The white dots represent items and enemy spawning locations.

Shooting is one of the most important skills in the game and it has a separate single task training environment. This map is a rectangular hall filled with a predefined number of enemies that roam across it but are not hostile towards the player. The player has a limited amount of ammo, enough for eliminating all the enemies if playing in a sensible way and must destroy as many enemies as it can. The episode ends when all enemies are gone, the player runs out of ammo or the timeout is reached. The agent gets a score of 100 for each eliminated enemy and a penalty of 10 for each shot it takes, with the final score being the sum of these.

The last but not least is the navigation skill. This subtask has the most complicated setting due to its more complex goal and reward function. The environment consists of two rooms that are not connected to each other and the agent is spawned randomly at one of the predefined locations. Each room has a different shape and has different number of obstacles in the form of walls and pillars. The score for this task is the same as the navigation module reward, but without living penalty.

There are two full task environments one that is trained upon and another that is used exclusively for evaluating the performance, i.e. investigating the generalization capabilities of the agent. These environments are referred to as "training map" and "validation map" respectively and their top view is shown in Figures 5 and 6 respectively. Notably, the validation map is only used in the final testing and it is not used for any parameter tuning nor for training. Both of these environments contain enemies and items, but in different amount and with different spawning schedules. The training map is the smallest of the two and the enemies are spawned six at a time, with the next batch spawning each time the agent manages to eliminate the current ones. The validation map is significantly larger and more complex. The enemies here are spawned in a different manner with first six enemies being spawned immediately at the beginning, after which a new enemy is spawned every

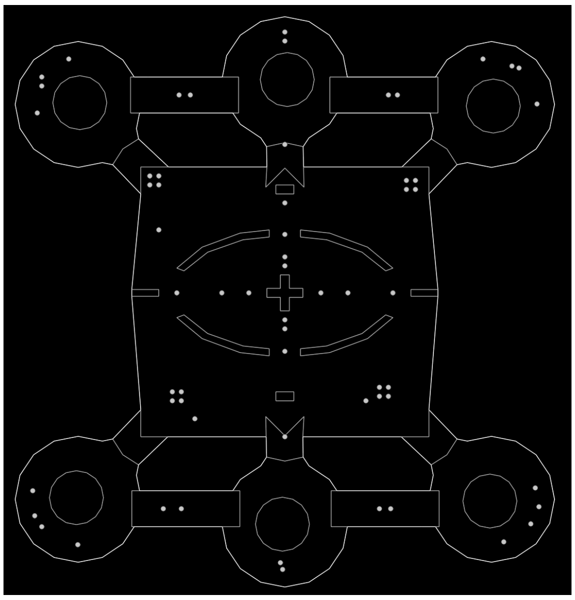

Figure 6: Full task validation map.

4.3 seconds. The score for these environments is the main task score, i.e. the number of the destroyed enemies multiplied by 100.

6.3 Training Procedure

In this subsection the details of the experiments are presented. Before running the final experiments, hyperparameters of the model were chosen via combination of manual experimentation and grid search over a subset of parameters. The weights of all modules were initialized to be vectors of ones before training.

Each experiment was run with 20 random seeds for a total of 3e6 learning steps corresponding to 9e6 in-game steps. Each step equals to three in-game frames, because frameskip of three frames was used for all of the final experiments. Frameskip was used, because it sped up the runs and skipping three frames did not negatively affect the performance during the preliminary testing. The performance of the agent was also evaluated by running the agent, with fixed weights, on the validation map for 50 episodes at several points in time during training roughly every 500 000 steps and before the start of training.

The full task only training procedure is quite simple. The agent is trained from scratch on the full task training map, also referred to as 'deathmatch practice', for all of the 3e6 steps. The learning rate α is linearly decreased from 0.5 to 0.001 during the duration of training, and exploration rate ϵ is linearly decreased from 0.3 to 0.01 during the first 2.5e6 steps and stayed constant afterwards.

Training with task rotation is similar to the full task only training except that instead of using only a single map, the agent is trained on multiple the training maps in the following sequence ['health gathering', 'ammo gathering', 'armor gathering',

’shooting practice’, ’navigation practice’, ’deathmatch practice’]. The ’deathmatch practice’ map is the same as the one used in the full task only method and all other maps are designed for single task learning. The agent is trained on each task for 10 episodes until switching to the next one. Learning rate and exploration rate are annealed with the same schedule as with full task only method. Importantly, the whole agent, i.e. all the modules, are working and learning on all of the maps simultaneously.

When pretraining is used, a separate single task agent is trained for each of the subtasks for 3e5 steps with learning rate linearly decreasing from 0.5 to 0.001 and exploration rate linearly decreasing from 0.5 to 0.01 for the first 2e5 steps and stayed constant for the rest of the training. During the single task training a constant feature was used, which allowed for learning a bias term, since it was observed to help with the training process. The constant feature allows for a single action to be learned when the feature vector is otherwise zero, i.e. when no objects of interest are present, and usually the agent learns to rotate itself until it finds an object of interest. This rudimentary seeking behavior is helpful in this case because unlike full agents the single task agents do not have map exploration module (unless that is the only module). After training on the single tasks, the full task agent’s weights are initialized with the learned weights and the agent is trained on the full task for additional 1.5e6 steps either on full task only or with task rotation. For both single map and task rotation settings the learning rate was decreased from 0.01 to 0 during the whole training and the exploration rate decreased 0.1 to 0.01 for the first 1e6 steps and kept constant afterwards. In total the pretrained agents see the same number of steps as those not using pretraining, i.e. $5 * 3e5 + 1.5e6 = 3e6$.

7 Results

In this section the results of the experiments will be presented. We will show the learning curves for all the training variants. The learning curves for training maps and tasks will be presented for the analysis of the individual tasks. The validation scores, i.e. the full task scores obtained on the validation map, will be used to help with comparison of the different training methods and are shown as Figure 15. All of the results are calculated over 50 runs with random seeds for each method. This makes the robustness of the approach clearer, but since learning is stochastic and can fall into the local minimum, the results may also contain very bad runs. For better understanding of the best-case performance, the maximal results over runs are presented as well. Additional videos of the trained agents in action can be found online [1].

[1] `https://helsinkifi-my.sharepoint.com/:f:/g/personal/ivkropot_ad_helsinki_fi/Ehy2yY8qjs9HkxP1NTQZy9oBlV_Na8CjSmZkRQzlXAAWdQ?e=AwBsgY`

7.1 Performance on the Training Environments

In this section we will present the learning curves that were achieved on the training environments for all of the training methods. These results show whether the agent was able to learn the task at all and how quickly.

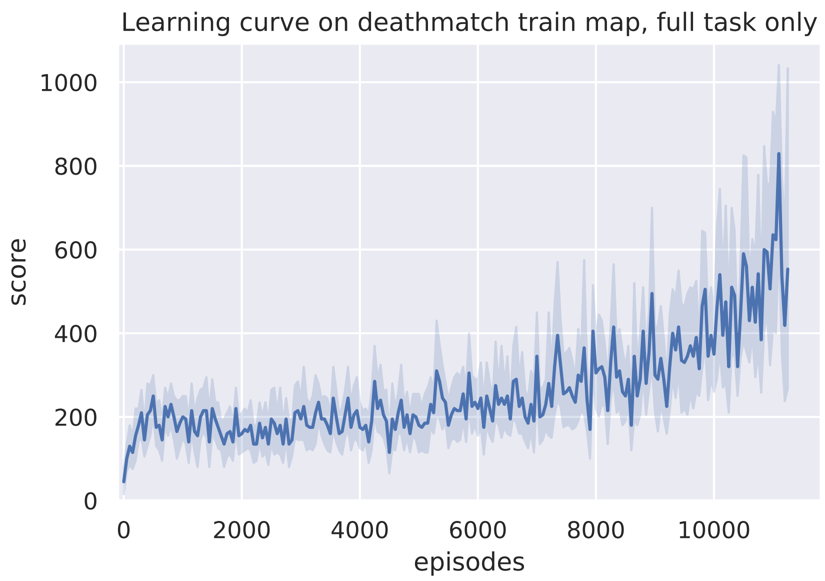

Figure 7: The average episode rewards of single task setting during training.

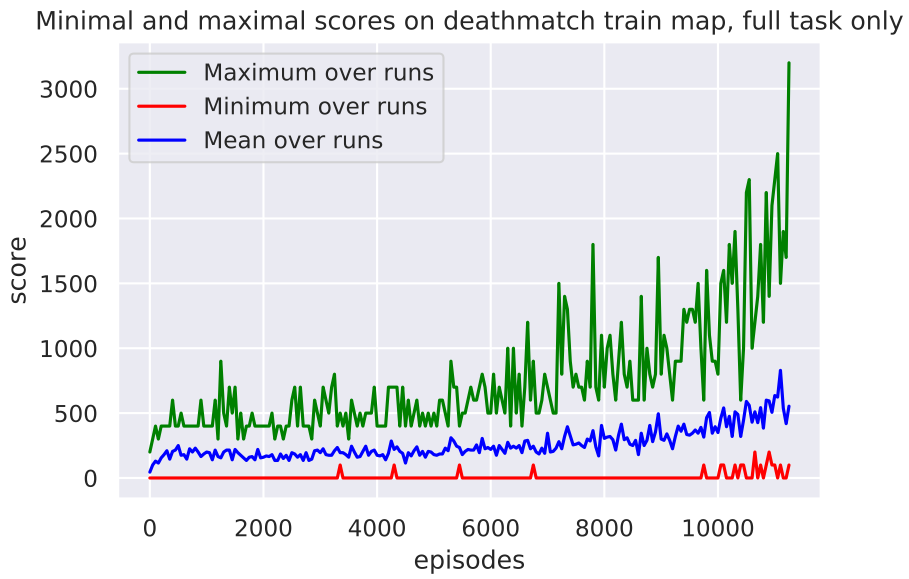

Figure 8: Minimal and maximal episode rewards of single task setting during training. The mean score, i.e. the blue line, is the same as the bold line in Figure 7.

Starting from the simplest training setting, which is training on the full task only from scratch, i.e. not using any pretrained weights, we can see the learning curve of the full task on the training map in Figure 7. From this curve we can see that the agent trained this way steadily improves its performance, even though it doesn't reach the highest rewards in general. Looking closer, a considerable variation in the scores is observed, which is even more clear in the Figure 8 that shows the highest and lowest scores obtained each episode across runs that use different random seeds. This graph also illustrates that while the average performance may be mediocre, some runs can get quite good results.

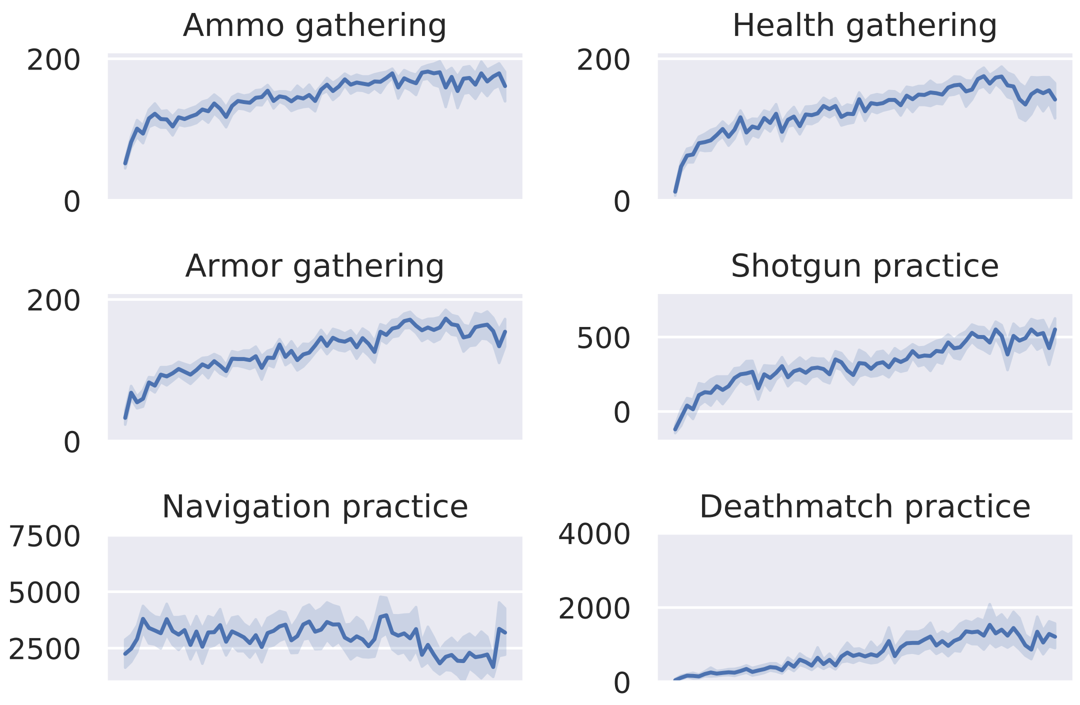

Figure 9: Learning curves during training for all tasks of task rotation. Bold lines indicate the mean and the shaded region the 95% confidence interval.

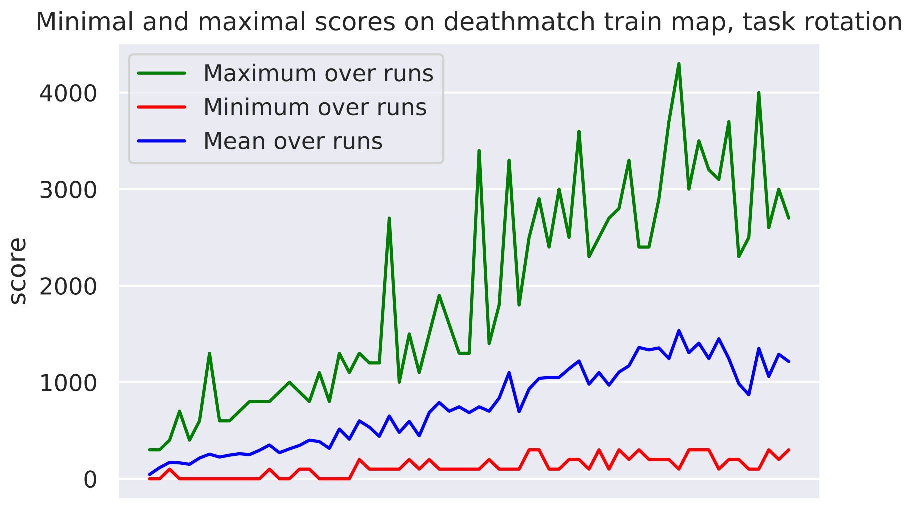

Figure 10: Minimal and maximal episode rewards of task rotation setting during training. The blue line is the same as the bold line in lower right of Figure 9.

Next is the case when the agent is trained without using pretrained weights but with task rotation, i.e. the agent is trained to solve individual subtasks on specially designed environments that are rotated in a sequence. Figure 9 shows the learning curves for each of the subtasks during training. From these graphs we can see that this training method also results in learning of the task. Moreover, we can also see that it learns each of the subtasks fairly well, especially considering that the theoretical maximum score in the item gathering tasks is 200 and in shooting practice it is 800. The Figure 10, which shows the full task train map performance in more detail, further confirms the fact that the agent learned a non-trivial policy. Moreover, we can also see that using task rotation favorably compares to the performance of an agent trained on full task only.

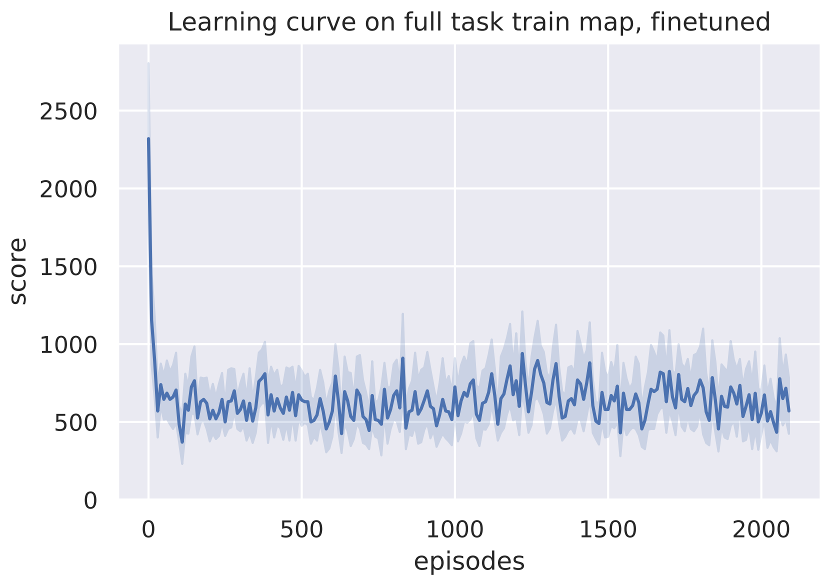

Figure 11: Learning curve of training on deathmatch task of an agent which whose weights were finetuned on full task only.

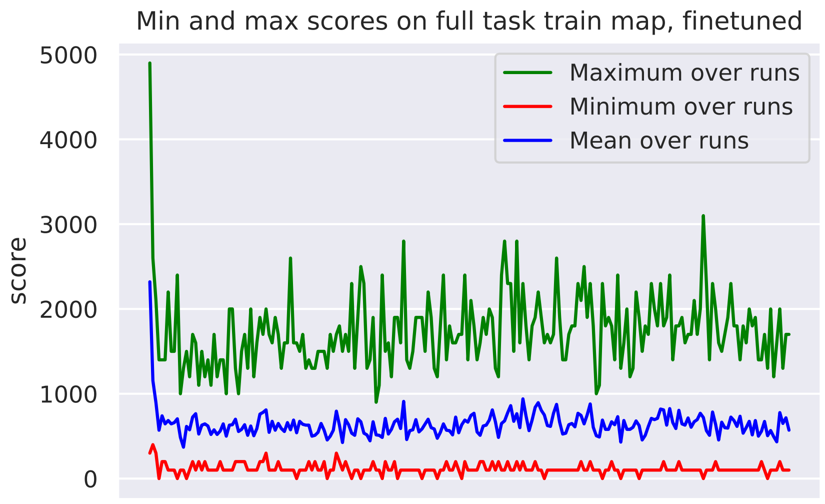

Figure 12: Minimal and maximal episode rewards during finetuning of an agent whose weights were finetuned on single task only.

Now that all the methods that were learning from scratch are covered, we turn to their variants that use pretraining, i.e. the weights for a module are initialized to those of an agent that was trained to solve the corresponding task. Beginning with an agent whose weights are finetuned on a full task training map only, the Figure 11 shows the learning curve on deathmatch task during the finetuning phase, and the Figure 12 shows the same data but with minimal and maximal values over runs explicitly stated. It can be seen that the performance of the agent during the finetuning phase immediately drops by a large amount, to the level similar to or even worse than without pretraining. The performance doesn't recover over the course of training, and by the end of finetuning phase the performance is nowhere close even to the starting point.

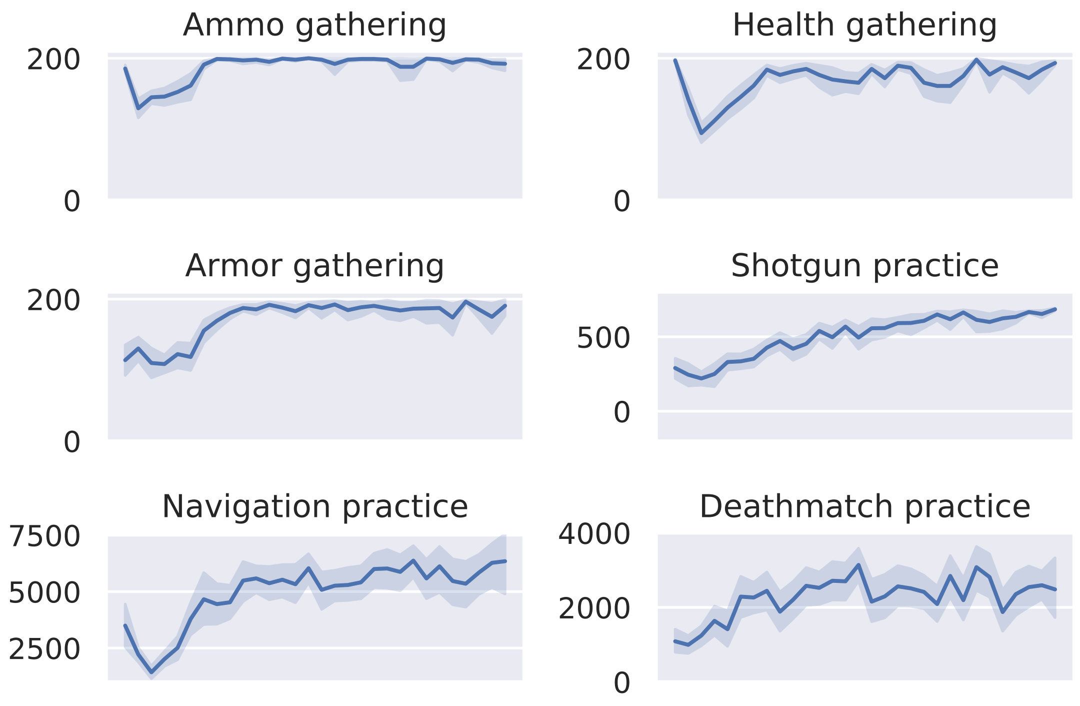

Figure 13: Learning curves during finetuning of pretrained agent for all the tasks of task rotation. Bold lines indicate the mean and the shaded region the 95% confidence interval.

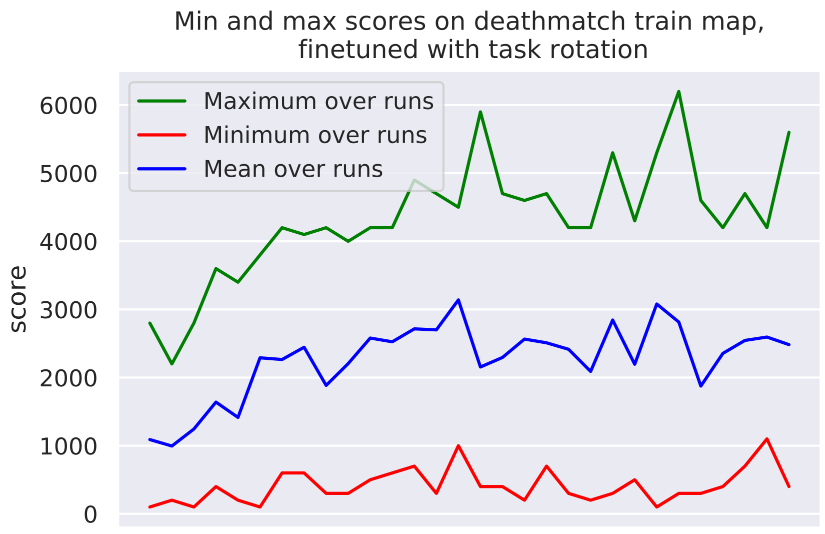

Figure 14: Minimal and maximal episode rewards during finetuning of an agent whose weights were finetuned with task rotation.

However, when finetuning a pretrained agent with task rotation, the picture changes significantly. As evidenced by Figures 13 and 14 the performance once again drops rapidly during the beginning of the finetuning phase. However, the performance quickly recovers and even gets better on some tasks than in the beginning of the training. Notably, the average performance on many of the subtasks, namely item gathering and shooting practice, gets very close to the best theoretical score, which means that the agent can solve these subtasks successfully almost always.

7.2 Performance on the Validation Environment

In this section the performance on full task validation environment is presented. These results show how well the agents are able to generalize to unseen environments, as well as allow for easier comparison between the training methods since all of them are run on the same validation environment with the same intervals between training steps. The validation score plots shown in this section also include the validation score for an agent, that uses pretrained weights but is not finetuned. This agent, surprisingly, achieves the best overall performance, which will be discussed further in Section 8.

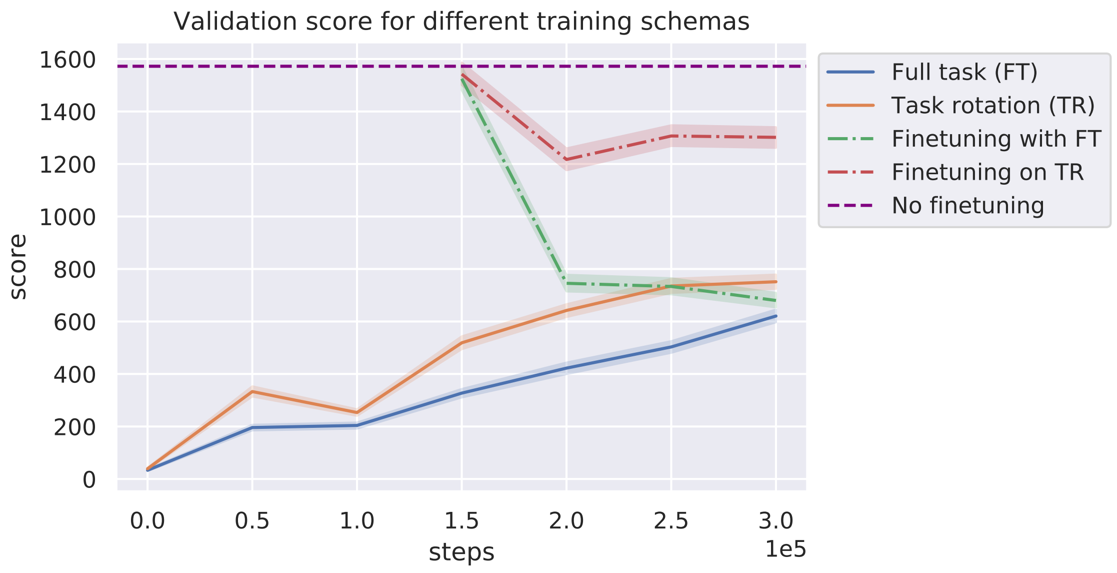

Figure 15: Validation results for all of the training setup variants. The purple line is the average validation reward over 1200 episodes for a pretrained agent that wasn't finetuned. The curves for the finetuning variants are offset for 1 million steps to accommodate for the learning used for pretraining the individual modules. Bold lines represent the mean and the shaded areas indicate the 95% confidence interval.

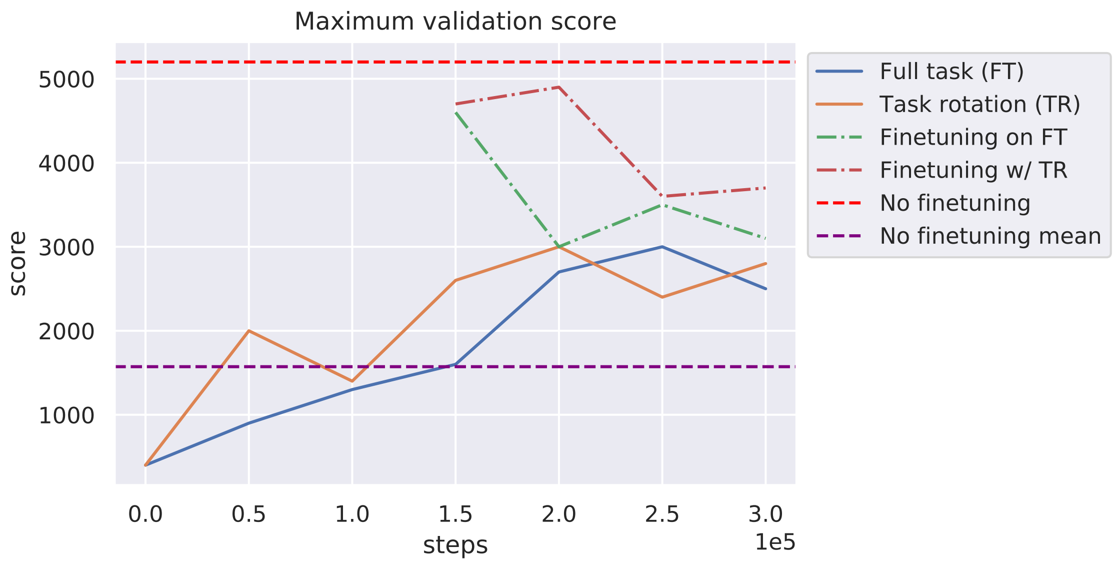

Figure 16: Maximum validation scores. The purple line, which is the same line as in Figure 15, is included for reference. The maximal score for an agent without finetuning is higher than the starting value of those with finetuning due to it being calculated over more episodes.

Starting again from the agent trained on full task only without pretraining, the performance of the agent on the validation map can be seen in Figure 15 in blue.

The average performance is indicated by the bold line and the shaded area is the 95% confidence interval. This curve shows that the agent trained on full task only in general achieves the worst performance amongst the training methods and has the slowest learning, though the final performance is close to all of the training methods except for finetuning with task rotation. Furthermore, the maximal validation map score is much larger than the average and is again quite close to the performance of two of the four other training methods as seen in Figure 16.

Continuing to validation results of training with task rotation from scratch we see from the Figure 15 that this method achieves better performance than both methods that use only a single map for training. Additionally, we also see that using task rotation results in faster learning. However, when looking at the maximal scores in Figure 16, the best scores at the end of the training are comparable to the best scores of the single map methods, which indicates that despite being better in general this method doesn't get that much better in the best case.

When looking at the performance of the methods that use warm start in Figure 15 we notice quite an interesting picture. When the finetuning is done using only a single map, the validation performance drops very much and doesn't recover at all. The final performance of the agent trained in this way is on par, or even worse, than training on from scratch. The maximal validation score of Figure 16 tells the same story.

Interestingly, when using task rotation for finetuning, the validation results look much better than finetuning without it. This training method still has a large drop in performance, though it is much smaller than when task rotation is not used. Even though the agent finetuned on task rotation doesn't recover to its starting point nor reaches the performance of purely pretrained agent, it still achieves very respectable scores and has the best overall performance, when the "no finetuning" case is ignored. The best runs, whose scores are shown in the Figure 16, mirror the results on average, by having a large drop in the beginning but still being better than other variants where the modules are trained together.

7.3 Visual Inspection of Trained Agent Performance

A visual inspection of the models was performed for all training methods. This inspection mostly confirmed the results presented in this section, i.e. that all of the methods resulted in learning, but some were better than others. However, the visual inspection allowed to identify some bizarre behaviors and failure modes.

Perhaps the most visible behavioral artifact that most of the agents exhibited was oscillatory behavior, where the agent could get stuck in one place choosing same, but opposite actions at each timestep, e.g. by continuously turning left and then right. This behavior was mostly observed with the worse performing agents, with the agent that was finetuned with task rotation showing only a little of this behavior and the pretrained agent didn't have this problem almost at all.

Getting stuck in general was quite common, especially among the worst performing agents, with them often getting stuck in corners or backing up to a wall. The weakly performing agents also seemed to sometimes develop some strange item picking behavior where they would try to approach them from a peculiar direction instead of just going straight towards it. Some seemingly random shooting was also observed in almost every configuration. None of the agents managed to come up with clever tactics and even the best performing ones would mostly just storm towards the enemy, sometimes dodging the enemy projectiles, though this could be attributed to random chance.

8 Analysis of Q-decomposition Performance

The results presented in the Section 7 paint a complex picture about the applicability and performance of the Q-decomposition method as applied in this work. On one hand, the agent learned to play quite well, especially considering how extremely simple the models used for learning the subtasks were. On the other hand, the design of the agent, in particular task decomposition and state engineering, required a considerable effort and, in the end, the modules pretrained on separate tasks outperformed all other methods using a fraction of their training time. All of these issues will be discussed in this section, starting with detailed discussion of the results and investigation of the finetuning performance droop. After this successes and failures of the trained models will be presented, and at the end general issues of applicability and working with Q-decomposition will be considered.

To make it easier to put the numbers into perspective, we can consider some example human scores, which were alluded to in the Section 7. The human scores mentioned in this work were collected from a sample size of only a couple of people in a weakly standardized way and are only intended to roughly illustrate the performance level that the agent achieves. A person, who has very little experience with video games, after being told the rules and given about 15 minutes to practice on the training map can achieve a score of up to 800 on the validation map, though most of the trials will be below it. In contrast, a person familiar with Doom, can easily achieve a score on the validation map of over 5000 and, with a bit of effort, over 10000, with possibly even higher scores achievable with more advanced tactics.

8.1 Performance Comparison of Different Training Methods

We will begin by comparing the different training methods between each other, which will give us understanding of the performance in general and highlight some interesting questions, which will be handled later in detail. Going through the methods in order of increasing performance following the validation scores we start with the simplest method where the agent is trained on a single map from scratch. This method has the slowest learning and its performance is among the lowest, but it still beats a novice player's performance, especially when looking at the maximal scores.

The next in line is training on a single map but starting with pretrained weights, which starts strongly but during finetuning quickly drops to the level of training on single map from scratch. This result was very surprising, since it was expected that the warm start would result in better performance. The finetuning case will be gone through in more detail later in this section.

Next comes the agent trained from scratch with task rotation that achieves better performance than training on the single map while also learning faster. This behavior was expected, since task rotation allows the agent to concentrate only on training a single module on each of the single task maps, which in turn eliminates a big part of the randomness and noise coming from other modules' action preferences, resulting in faster learning of individual subtasks.

The benefits of task rotation are clearly seen in the second-best performing training method which is finetuning with task rotation. While this method also shows a large drop in average performance, it nevertheless gets average score not far from the best. Furthermore, the maximal score is, arguably, very impressive since it doesn't degrade and at the mark of 2.5e6 steps achieves a score that is very close to the highest score achieved by any of the methods. This is a very good result considering that the agent was evaluated less times at that point than the pretrained only agent (red line in Figure 16). The fact that this method achieved such a performance can again be attributed to the task rotation method giving the agent chance to concentrate on a single task and reducing the noise in action selection. This claim will be investigated in more detail further in this section together with closer look on finetuning.

Finally, the best performing, and perhaps most surprising, agent is the one that simply consists of pretrained modules and is not finetuned in any way. This agent not only achieves the best scores, but it also seems to be the most stable during the visual inspection of its gameplay, with the least amount of irrational behavior, like oscillation, and getting stuck. This aversion to getting stuck is perhaps the main reason that the average score of this agent is higher than the one finetuned on task rotation, which demonstrates such behavior. The main thing that makes this result so unexpected is the fact that all the modules were trained one by one, which means that the behavior policy they trained on didn't take any other modules into account. However, the point in using the Sarsa based Q-decomposition algorithm was the claim that the modules which assuming total control over the action selection, as is the case with modules trained in isolation, lead to suboptimal policies, i.e. the illusion of control [28].

8.2 Analysis of the Performance Drop Caused by Finetuning

It is a very interesting question as to why the agent with pretrained modules performs better than any real Q-decomposition agent in spite of the illusion of control. This behavior is most likely due to a combination of factors.

One element contributing to the issue, is purely due to the proposed task decomposition and state representation. The main task is decomposed into several smaller tasks, like individual item gathering, shooting and exploring the map. However, while all of these subtasks are essential parts of the main task, they often don't need to be solved at the same time, in a way making the subtasks independent. The proposed reward function structure makes concentrating on one module even easier, since e.g. the action-values of shooting module dominate other modules, health gathering dominates exploring, etc. This means that, for example, when no items or enemies are present, the agent only explores, or if health, armor and ammo are full but there are enemies in view, the agent will concentrate on shooting, since it always dominates on navigation module. So, in the end there aren't that many occasions when the agent needs to learn a useful compromise which in part leads to the good performance of individually trained modules. As a side note, while the performance of the pretrained agent is the best, the visual inspection of the gameplay seems to suggest that the finetuned agent (with task rotation) behaves in a bit smoother and more natural way as opposed to quite sharp actions of the pretrained agent.

Another component that contributes to the surprisingly good performance of the pretrained modules agent is also the one that is the most likely the cause of the sudden drop in performance during finetuning. This problem seems to arise from the chosen function approximation method and the state representation, and in short it manifests itself in inability to learn well in the presence of noisy action selection. This also largely explains why finetuning with task rotation resulted in much smaller mean performance drop, which is because during task rotation most of the time the agent is essentially only using one module, excluding navigation module, for action selection and thus having very little noise in action selection.

The effect of noise on the performance is illustrated in Figure 17 that shows the average validation map score for an agent whose modules are initialized to pretrained weights, except that shooting module is finetuned by its own on a shooting practice map with different values of exploration parameter ϵ, i.e. the probability of making a random action. The shooting module was chosen to be the subject of finetuning, because it was observed that finetuning it or the exploration module had the largest impact on the performance. As can be seen in that figure, the larger the amount of random actions in the agents behavior policy, the lower is the final reward, which seems to indicate that the noise in the action selection is the main culprit. Incidentally, during a normal finetuning run the shooting module is not obeyed about 10% of times it has an action to offer.

Digging deeper, we can also look at the learned weights with different configurations in Figure 18. In this figure we can see that shooting modules weights after finetuning with all modules (Figure 18b) and finetuned alone (Figure 18c) look quite similar, with both being more spread than the weights immediately after the pretraining 18a. Furthermore, the larger the ϵ the more the weight mass is spread.

All this suggests that the chosen function approximation and state representation may be unable to learn well in the presence of the noisy action selection, thus being unable to take full advantage of the Q-decomposition. In essence what is happening is that the agent still learns to solve its subtask, it just tries to compensate for the actions taken following other modules, which are seemingly random from its point of view. This is perhaps also one of the drawbacks of the Q-decomposition approach, since the presence and intent of other modules is signaled to the modules implicitly.

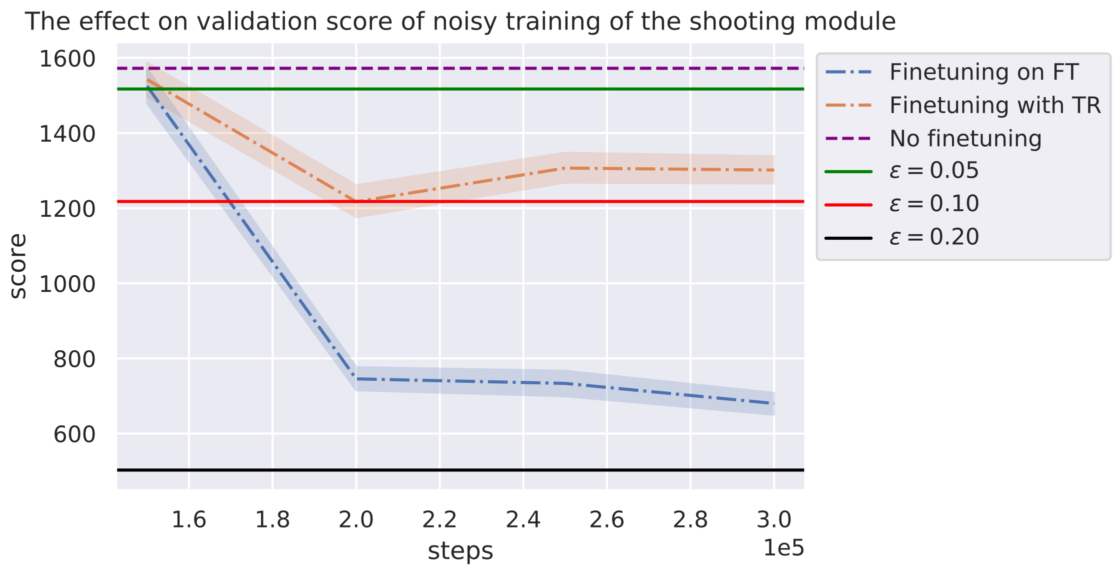

Figure 17: Mean validation map scores for an agent consisting of pretrained modules that where not finetuned with exception of shooting module. The shooting module was trained starting from pretrained weights by itself on a shooting practice map with different possibilities of random actions $\epsilon \in \{0.05, 0.10, 0.20\}$.

It needs to be noted that the explanations provided above are at this point practically hypotheses and truly confirming them would need more experiments involving multiple models, algorithms and, ideally, different tasks, which is beyond the scope of this work. To this end, here are some other possible explanations, that were considered but deemed highly unlikely to be the cause. The most obvious explanation would be, of course, wrong hyperparameters, since not all of the possible combinations were tried out due to limited time and their large number. However, a lot of the sensible hyperparameters were tested, both manually and using grid search, so that we can dismiss this hypothesis as unlikely. Another possible reason could be related to the reward function, since maybe the agent does actually find the optimum for the given reward function. This explanation is unlikely as well, because different sensible reward functions were tried and all of them showed the degradation of the agent's performance during finetuning.

8.3 Shortcomings of the Proposed Agent

Now that the results have been gone through, we can say that the presented agents that use Q-decomposition succeeded in learning to play better than a complete

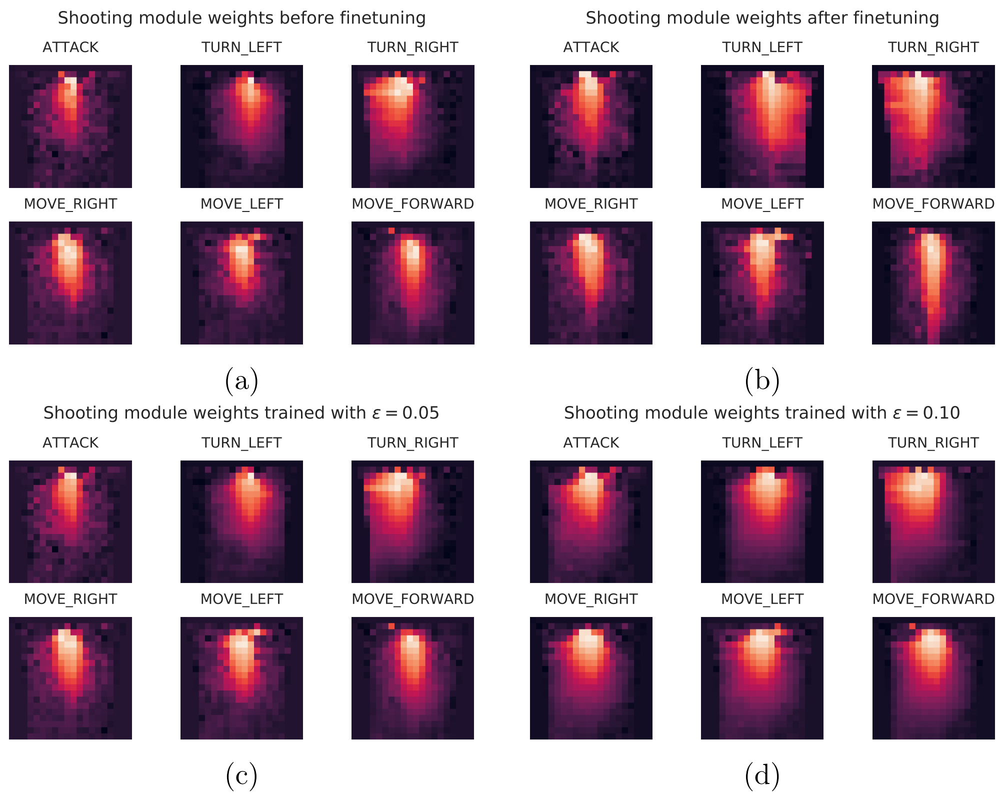

Figure 18: Weights of the shooting module before finetuning and after finetuning in different ways. The difference in vertical spread between (b) and others is due to them being trained/finetuned on different maps, whose sizes were different.

novice or baked in game AI, using all of the training methods, which is undoubtedly a success. However, even the best of the agents fails to get to the "expert" human level both in maximum score and in average.

There are a few reasons for the presented agents' inability to reach the heights of good human players, one of which is the issues with learning with noisy action selection covered before. Another factor, which is perhaps even more important, is the extreme simplicity of the used models. This simplicity mainly comes from the usage of linear function approximation, which greatly reduces the ability of exploiting interaction between state features. Furthermore, the state representation used doesn't have any memory, meaning that anything it doesn't immediately see doesn't exist making the states non-Markovian as well, since the state doesn't have all the needed information. Moreover, the agent doesn't build nor use any models. Finally, the Q-decomposition algorithm that was used for enabling task decomposition itself is a very simple approach, in essence being just a fancy version of Sarsa, that doesn't provide much help for the learning of the modules together.

Of course, the goal of this work was to design components that use the simplest possible methods to achieve a goal together that would not be possible by these methods individually. This goal was indeed achieved. However, playing Doom well, i.e. on human level, requires, among others, remembering item positions and building models of one's surroundings and enemies. Human players use these techniques, while it is practically impossible for the proposed agent to utilize this knowledge. In the simplest case this is seen when the agent detects an enemy in the corner of its view and at the next step loses it from its field of view. On the next step it wouldn't turn towards that enemy since it doesn't exist anymore from the perspective of the agent, while a human player would remember everything and act correspondingly. All of this taken together makes it unreasonable to expect the proposed agent to reach "expert" level performance, while making its final performance of beating "novice" performance and even matching average run of "expert" look somewhat even more impressive.

8.4 Practical Considerations When Using Q-decomposition

The last part of the discussion will be about using Q-decomposition for task solving in general. The promise of using modular reinforcement learning is in the ability of this technique to reduce problems with large state spaces into more manageable parts, while being an obvious choice for a large number of natural tasks. This in turn should allow us to use simpler models and less computation, make designing the reward a bit easier and the engineering of state representation should also become simpler.

All in all, Q-decomposition did deliver on its promises when applied to the simplified Doom task of this work. It allowed us to use extremely simple models to learn a task that would be practically impossible for such methods but became possible after decomposing the main task into natural subtasks.

However, this approach is in no way a silver bullet that is able to solve any problem without additional effort from the designer. Deciding on the suitable task decomposition and creating sensible reward functions took a surprisingly long time, and it is easy to make a wrong decision at this level. Furthermore, even with simple subtasks designing a good state representation is still quite an undertaking. Most importantly, even when all the components, decomposition, reward function, state representations and function approximation, are chosen and are looking like a sensible choice, it is still not guaranteed that they will behave well when working together during learning and take full advantage of Q-decomposition, as evidenced by discussion earlier in this section.

To summarize, MRL is in practice a very powerful and promising approach, and Q-decomposition in particular does seem to work even on complex problems like Doom. However, there are a lot of hidden considerations and Q-decomposition might be a bit too simple of an approach to become widely useful.

9 Conclusion and Future Work

In this thesis we presented an agent for playing a simplified version of Doom that was built using Q-decomposition [28] and linear function approximation. Together with the agent, three methods of training such an agent were presented and evaluated. The results indicate that the proposed agent was able to learn how to play Doom, trained with any of the proposed training methods. However, it turned out that the best performance was actually achieved with an agent whose modules were simply pretrained individually on specially designed training environments and combined without further learning together. This was likely the result of the usage of linear function approximation, which indicated difficulties with learning in presence of multiple competing modules.

There are many possible directions for future work. The most obvious direction would be to just add more modules to the current agent. For example, a module that tries to avoid enemy projectiles would probably improve the performance quite a lot.

Another easy target would be to try out the similar agent as presented here, but with another function approximation methods, such as ANNs, and possibly another state representation. This could also shed some light on the limitation of LFA used in this work when combined with Q-decomposition.

It would be also interesting to apply, compare and potentially combine some other MRL approaches such as W-learning [12], Arbi-Q [21, 31] or, since it seems that it is quite possible to separate the subtasks very well, even GM-Q [12, 16].

Promising direction from the perspective of applying this method to real Doom, would be to apply machine vision approaches for extraction of the object information, instead of relying on ground truth provided by the ViZDoom environment, and adding map information e.g. with SLAM [5].

Another direction for future work would be exploration of this problem from the point of view of MORL approaches, for example by learning policies so that the agent's policy could be changed during the acting phase through changes to the priority of subtasks, e.g. by making health gathering more important when the agent is low on health. Parallel to exploration of other MORL/MRL methods, it could be quite fruitful to do a further study of possibilities of alternative training methods for modular agents, e.g. by using adaptive training to detect a failure of a particular module or task to learn and training/retraining it online.

References

1 Asperti, A., De Pieri, C., and Pedrini, G. Rogueinabox: an environment for roguelike learning. *International Journal of Computers 2* (2017).

2 Astrom, K. Optimal control of markov processes with incomplete state infor-

mation. *Journal of Mathematical Analysis and Applications 10*, 1 (1965), 174 – 205.

3 Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. Deepmind lab. *CoRR abs/1612.03801* (2016).

4 Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (jun 2013), 253–279.

5 Bhatti, S., Desmaison, A., Miksik, O., Nardelli, N., Siddharth, N., and Torr, P. H. S. Playing doom with slam-augmented deep reinforcement learning. *CoRR abs/1612.00380* (2016).

6 Campbell, J. C., and Verbrugge, C. Learning combat in nethack. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17), October 5-9, 2017, Snowbird, Little Cottonwood Canyon, Utah, USA.* (2017), B. Magerko and J. P. Rowe, Eds., AAAI Press, pp. 16–22.

7 Cerny, V., and Dechterenko, F. Rogue-like games as a playground for artificial intelligence–evolutionary approach. In *International Conference on Entertainment Computing* (2015), Springer, pp. 261–271.

8 Geramifard, A., Walsh, T. J., Tellex, S., Chowdhary, G., Roy, N., and How, J. P. A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Foundations and Trends in Machine Learning 6*, 4 (2013), 375–451.

9 Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)* (2017), IEEE, pp. 3389–3396.

10 Higgins, I., Pal, A., Rusu, A. A., Matthey, L., Burgess, C., Pritzel, A., Botvinick, M., Blundell, C., and Lerchner, A. DARLA: improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 1480–1490.

11 Howard, R. A. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

12 Humphrys, M. Action selection methods using reinforcement learning. *From Animals to Animats 4* (1996), 135–144.

13 Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K., and Graepel, T. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science 364*, 6443 (2019), 859–865.

14 Juliani, A., Khalifa, A., Berges, V., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., and Lange, D. Obstacle tower: A generalization challenge in vision, control, and planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019* (2019), S. Kraus, Ed., ijcai.org, pp. 2684–2691.

15 Kanagawa, Y., and Kaneko, T. Rogue-gym: A new challenge for generalization in reinforcement learning. *CoRR abs/1904.08129* (2019).

16 Karlsson, J. *Learning to Solve Multiple Goals.* PhD thesis, University of Rochester, 1997.

17 Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games* (Santorini, Greece, Sep 2016), IEEE, pp. 341–348. The best paper award.

18 Konidaris, G., Osentoski, S., and Thomas, P. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence* (2011), AAAI 11, AAAI Press, pp. 380–385.

19 Lample, G., and Chaplot, D. S. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA* (2017), S. P. Singh and S. Markovitch, Eds., AAAI Press, pp. 2140–2146.

20 Liang, Y., Machado, M. C., Talvitie, E., and Bowling, M. H. State of the art control of atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016* (2016), C. M. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, Eds., ACM, pp. 485–493.

21 Lin, L. J. Scaling up reinforcement learning for robot control. In *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993* (1993), P. E. Utgoff, Ed., Morgan Kaufmann, pp. 182–189.

22 Liu, C., Xu, X., and Hu, D. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Trans. Systems, Man, and Cybernetics: Systems 45*, 3 (2015), 385–398.

23 Mauldin, M. L., Jacobson, G., Appel, A. W., and Hamey, L. G. C. Rog-o-matic : a belligerent expert system.

24 Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (2016), ICML16, JMLR.org, pp. 1928–1937.

25 Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (Feb. 2015), 529–533.

26 Roijers, D. M., Vamplew, P., Whiteson, S., and Dazeley, R. A survey of multi-objective sequential decision-making. *J. Artif. Intell. Res. 48* (2013), 67–113.

27 Rummery, G. A., and Niranjan, M. On-line q-learning using connectionist systems. Tech. rep., 1994.

28 Russell, S., and Zimdars, A. L. Q-decomposition for reinforcement learning agents. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning* (2003), ICML'03, AAAI Press, pp. 656–663.

29 Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

30 Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature 550* (Oct. 2017), 354–.

31 Simpkins, C. L., and Jr., C. L. I. Composable modular reinforcement learning. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.* (2019), AAAI Press, pp. 4975–4982.

32 Sprague, N., and Ballard, D. Multiple-goal reinforcement learning with modular sarsa(o). In *Proceedings of the 18th International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 2003), IJCAI'03, Morgan Kaufmann Publishers Inc., pp. 1445–1447.

33 Sutton, R., and Barto, A. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning series. MIT Press, 2018.

34 Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell. 112*, 1-2 (Aug. 1999), 181–211.

35 Tesauro, G. Practical issues in temporal difference learning. *Mach. Learn. 8* (1992), 257–277.

36 Tesauro, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6*, 2 (1994), 215–219.

37 Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems* (2017), pp. 5392–5402.

38 Watkins, C. J. C. H. *Learning from Delayed Rewards.* PhD thesis, King's College, Oxford, 1989.

39 Watkins, C. J. C. H., and Dayan, P. Technical note q-learning. *Mach. Learn. 8* (1992), 279–292.

40 Wu, Y., and Tian, Y. Training agent for first-person shooter game with actor-critic curriculum learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (2017), OpenReview.net.

41 Wydmuch, M., Kempka, M., and Jaśkowski, W. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games* (2018).