Hierarchical Reinforcement Learning for Playing a Dynamic Dungeon Crawler Game

Remi Niël
Department of Artificial Intelligence
Bernoulli Institute
University of Groningen
Groningen, The Netherlands
r.f.niel@student.rug.nl

Marco A. Wiering
Department of Artificial Intelligence
Bernoulli Institute
University of Groningen
Groningen, The Netherlands
m.a.wiering@rug.nl

***Abstract*—This paper describes a novel hierarchical reinforcement learning (HRL) algorithm for training an autonomous agent to play a dungeon crawler game. As opposed to most previous HRL frameworks, the proposed HRL system does not contain complex actions that take multiple time steps. Instead there is a hierarchy of behaviours which can either execute an action or delegate the decision to a sub-behaviour lower in the hierarchy. The actions or sub-behaviours are chosen by learning the estimated cumulative reward. Since each action only takes one time step and the system starts at the top of the hierarchy at every time step, the system is able to dynamically react to changes in its environment. The developed dungeon crawler game requires the agent to take keys, open doors, and go to the exit while evading or fighting with enemy units. Based on these tasks, behaviours are constructed and trained with a combination of multi-layer perceptrons and Q-learning. The system also uses a kind of multi-objective learning that allows multiple parts of the hierarchy to simultaneously learn from a chosen action using their own reward function. The performance of the system is compared to an agent using MaxQ-learning that shares a similar overall design. The results show that the proposed dynamic HRL (dHRL) system yields much higher scores and win rates in different game levels and is able to learn to perform very well with only 500 training games.**

***Index Terms*—Hierarchical reinforcement learning, Games, Multi-layer perceptron, Q-learning**

I. Introduction

Reinforcement learning algorithms enable an agent to learn to optimize its behaviour from receiving rewards from an environment, with which the agent interacts. Games are a thriving area for reinforcement learning (RL) and have been so for a long time [1], [2]. They provide varying challenges that produce situations in which new and improved RL algorithms can show their strengths and weaknesses. A relatively recent example is double Q-Learning as a new deep reinforcement learning algorithm for playing Atari 2600 games [3], and a classical example would be TD-Gammon [4] that learned to play the game Backgammon at human expert level. Among RL techniques applied to games, temporal-difference learning variants and evolutionary RL [5] are the most popular ones [1]. A lot of RL techniques have in common that they are based on a Markov decision process (MDP). An MDP is a sequential decision making problem for fully observable worlds where the Markov property is assumed [6]. The Markov property holds if the next state can be predicted using only the information that is available in the current state observation of the agent, which means that the previous interaction history does not need to be used for decision making.

Hierarchical reinforcement learning (HRL) is an extension of RL where problems are decomposed into smaller sub-problems, which allows RL to scale up to more complex problems [7]. Dungeon crawlers and most other action games contain many sub-tasks which have to be performed either in parallel or in sequence. HRL allows these sub-tasks to be based on their own MDP, which results in a divide and conquer strategy. Some examples of HRL are MaxQ-learning [8], the options framework [9] and Q-decomposition [10].

In Q-decomposition an agent is built from simpler sub-agents and each of the sub-agents runs its own learning process. The complex agent has a central arbitrator, which chooses the action that has the highest sum of expected rewards given by the sub-agents.

In the options framework [9], options are introduced, where options are complex sets of actions which take multiple time steps to complete. An agent using the options framework can select such an option which would run until a terminating state of the option is encountered. The options framework can be extended by adding the possibility to terminate outside of terminating states. The early termination could for example occur if the option takes too much time. This can prevent the overall system's behaviour to get stuck in an option which has become sub-optimal. A disadvantage is that the programmer has to determine when an option can terminate early.

In MaxQ-learning [8], an MDP is decomposed into a hierarchy of smaller semi-MDPs (SMDPs), where the value function of the decomposed MDP is the additive combination of the value functions of the smaller SMDPs [8]. A hierarchy of behaviours is then constructed which corresponds with the SMDPs. These behaviours can either execute simple actions or execute other behaviours which take multiple time steps. The tasks of the behaviours depend on their location in the hierarchy. The top of the hierarchy is responsible for deciding the current sub-goal, while behaviours lower in the hierarchy are responsible for attaining their sub-goals. A disadvantage of this system is that the current sub-goal cannot be changed until a behaviour that is currently running has reached a terminating state. From this it follows that either the system can not react quickly to sudden changes in the environment or that the programmer has to generate extensive terminating rules for every behaviour in the hierarchy.

Related to HRL is the recent Hybrid Reward Architecture (HRA) [11]. Here the reward function of an environment is decomposed into multiple different reward functions. Each of them is assigned to its own RL agent. Each agent gives its expected value for every possible action to an aggregator, which combines those into a single value for each action. The current action is then selected based on that aggregated value. HRA was used to learn to optimally play the game Ms. Pac-Man in [11].

Contributions We propose a new approach to HRL in which an MDP is decomposed into MDPs. Each time the system needs to determine which action to take, it goes through the entire hierarchy and ends up with a single primitive action. The result of this is that choices by nodes in the hierarchy always take exactly one time step which is why MDPs still apply. The system can continue its sub-behaviours multiple time steps if the choices of the hierarchy result in the same leaf of the hierarchy. Hence sub-behaviours can run as long as they are needed, but can also stop at any point in time should some other sub-behaviour be better suited for the current situation. The system can react dynamically to changes in its environment and instead of the programmer determining when that happens, the system learns this by itself. To speed up learning all behaviours in the hierarchy are trained simultaneously on an action if the following two conditions hold: 1) the behaviour can choose the performed action, and 2) the behaviour's overall goal is currently reachable.

For learning to play the dungeon crawler game, the HRL system is combined with multi-layer perceptrons (MLPs) and Q-learning [12] to learn the value function for all behaviours. The combination of RL and MLPs has been successfully applied to other games [13]–[15]. The MLPs receive higher-order inputs, an approach where only a subset of (processed) inputs is used. This approach has been successfully applied to the game Ms. Pac-Man where it learned to play the game very well within 10,000 training games [14].

A dungeon crawler game was developed in such a way that the agents can be tested on both simple and complex tasks. Levels consist of one or more rooms separated by doors. One of the rooms contains an exit which has to be reached by the agent in order to win. Doors are locked and can be unlocked using keys which have to be picked up by the agent. Rooms can also contain spawners, these spawners spawn enemy units which attack the agent. The agent can move and shoot in 8 directions, resulting in 16 different possible actions.

The performance of the proposed dHRL system will be compared to that of an agent using MaxQ-learning in varying levels: a simple level where only navigation needs to be learned, a small level with enemies and four rooms, and finally a large level with many rooms and more enemies. This will show how the systems compare to each other in environments of varying complexity.

Outline Section II describes the game and the theory behind the used RL algorithms. Section III explains the HRL algorithms and the constructed behaviour hierarchies for the agents. Section IV describes the performed experiments and their results. Section V presents our conclusions.

II. Methods

A. Dungeon Crawler

Our game is a dungeon crawler modelled to be a simplified version of the game "Gauntlet" on the NES (1988). The goal of the game is to reach the exit of a level, while most of the time the path is blocked by locked doors and enemies. Points can be gained by reaching the exit, picking up keys, opening doors and destroying enemies, see Figure 3 for an illustration of this type of game.

The agent is controlled by either manual input or an artificial intelligence system (AI). The AI can directly access all important information: where are keys, doors, enemies, etc. Note that the AI can only access information which a human player could see or deduce while playing. The AI also has access to a path-finding algorithm that determines the travel distance between two points using the A* algorithm.

There are 2 different types of enemies: spawners and ghosts. Ghosts are simple enemies, when the player is in range they walk straight to the player, and the ghosts die when they contact the player. When this happens the player gets damaged and if a player's health reaches 0, the player dies and the game ends. Spawners are stationary and do not attack the player but spawn ghosts. The player has 16 possible actions it can execute any moment: shoot or move in eight directions. When the player shoots, the bullet continues and kills all enemies (including the spawner) until a wall is hit. The player's score determines its performance, the score function can be found in Table I.

B. Reinforcement Learning

We use reinforcement learning to teach a hierarchy of neural networks to play the game. For a reinforcement learning system you need 5 parts, a model, an agent, actions, a reward function and a value function [16]. In our case the game is the model, the agent consists of a hierarchy of neural networks, the policy determines how states are mapped to actions using the value function, the reward function defines which game events get rewarded or punished, and finally the value function modelled with the neural networks returns the expected sum of discounted future rewards for state-action pairs. The goal of the agent is to maximise the sum of rewards.

TABLE I: Score gained for specific events

Event	Score
Damaged enemy	10
Damaged player	-10
Picked up key	5
Opened door	10
Reached exit	100
Impossible move	-0.1

TABLE II: Game states are represented by 75 input variables

Input type	Amount
Agent health	1
Keys	1
Normalised distance to target compared to other targets	3
Normalised distance to specific targets from tiles adjacent to agent	4 for each target type (12 in total)
Amount of enemies that can be hit in a specific direction	8
Vision grid	49
Normalised sum of enemies in shooting range or close to the agent	1

C. Q-learning

Reinforcement learning algorithms are used to optimize the action-selection policy by letting the agent interact with the environment. From this interaction, the agent observes which state-action pairs are visited and which rewards they produce. There are various learning algorithms that can be used to learn the value function. We use Q-learning here, because previous research indicates it works well for similar problems [14], [15].

A state at time t is referred to as s_t and an executed action at time t is denoted as a_t. The total reward received after action a_t and before s_{t+1} is denoted as r_t. For HRL, the time that r_t spans can be arbitrarily long. Q-learning is an online learning algorithm that uses its experiences in the form of (s_t, a_t, r_t, s_{t+1}) to update the state-action value function. The state-action value function estimates the expected sum of future rewards given a state and an action. The general Q-learning rule is [12]:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q_t(s_t, a_t))$$

Where α is the learning rate and γ is the discount factor. The learning rate determines how strongly the error changes the existing value function, while the discount factor determines the importance of future rewards vs immediate rewards.

The neural networks are updated using the back-propagation algorithm where the target value is determined by the Q-learning algorithm. In case the last action was a terminating action the final reward gained is used as target for the last state-action pair:

$$T(s_t, a_t) = r_t$$

In all other situations the following formula is used:

$$T(s_t, a_t) = r_t + \gamma \cdot \underset{a}{max}\, Q(s_{t+1}, a)$$

After computing the target for the multi-layer perceptron, the corresponding action a_t is trained with the target value using the state representation of s_t as input. We use MLPs with outputs that represent the Q-values of all possible actions or sub-behaviours.

III. Hierarchical Reinforcement Learning

In an attempt to reduce the overall complexity of the system, HRL systems decompose large MDPs in hierarchies of smaller (s)MDPs. It is a divide and conquer strategy where a difficult task is divided into a set of sub-tasks which are easier to solve. The bottom of the hierarchy contains simple tasks which directly call primitive actions, while tasks above determine which sub-task(s) should currently be running. We implemented two different HRL systems and compared their results: MaxQ-learning [8] and our novel dynamic HRL (dHRL).

Some properties are shared between the systems in the experiments. In both systems the nodes in the hierarchy use MLPs to approximate their value function. They also use Boltzmann exploration during training, which is used on the output of the MLPs. Boltzmann exploration gives actions (or sub-behaviours) with high expected values a larger chance to be selected. The temperature changes the absolute difference in chances, high temperatures reduce the difference while low temperatures increase the difference. The formula describing this exploration strategy is:

$$P(a|s) = \frac{exp(Q(s,a)/T)}{\sum\limits_{a' \in A} exp(Q(s,a')/T)}$$

Where $P(a|s)$ is the probability an action is chosen given the current state, T is the temperature and A is the set of possible actions.

Both systems also use the same global state representation, from which each behaviour within the hierarchy uses a subset as its input. The global state representation consists of 75 inputs, an overview of which can be found in Table II. One of the inputs simply contains the amount of keys the player currently has, another the amount of health normalised to a value between 0 and 1.

The next 15 inputs are used to give the agent information about the relative distances to the closest key, door and exit. These distances are calculated using the A* algorithm. The distances are then normalised between 0 and 1, where 0 means furthest away and 1 closest, with a possible -1 value if the target is not reachable. From the 15 inputs, 3 inputs are used to represent the relative distances to the targets compared to each other. These are normalised with respect to each other. The other 12 distance inputs are divided up into three groups of four, one group for each target. The four distances are between the four tiles adjacent to the player and one of the three targets.

The values are normalised within the groups as mentioned before.

The rest of the inputs are used to represent enemies. A vision grid is included covering a 7×7 area around the agent, which gives a 1 if there is an enemy on that tile and 0 if there is not. Then for each direction in which the agent can shoot the amount of enemies that would be hit if the agent were to shoot in that direction is returned. Finally there is a safety input which has the value $\frac{1}{1+x}$, where x is the sum of all enemies within the 7×7 square and the enemies which the AI could hit if it shoots.

A. MaxQ-learning

In MaxQ-learning the programmer constructs the hierarchy of sub-tasks. For every task the programmer determines the rewards, when rewards are received and the terminating states. We implemented the MaxQ-Q algorithm [8] in which each sub-task has two types of rewards: pseudo and normal rewards. Pseudo-rewards are used to train only the specific sub-task, while normal rewards are also passed on to the tasks higher up in the hierarchy.

There are two different types of actions a task-module (or behaviour) can perform: primitive actions and starting other sub-task behaviours. Primitive actions take one time step and are the most basic actions the agent can perform, in our case these are moving and shooting.

The algorithm will attempt to choose optimal sub-behaviours until it reaches a leaf in the hierarchy. It then keeps running that behaviour until one or more nodes in its hierarchy path reach a terminating state. After this it will update and terminate all behaviours underneath the highest node in the hierarchy that reached a terminating state.

When a behaviour is started it keeps track of the discounted sum of rewards. When the behaviour finishes it passes the discounted sum of rewards onto the behaviour that started it. A behaviour updates its network using the Q-learning rule where the reward used is sum of the normal and pseudo rewards gained during the chosen behaviour plus the discounted sum of normal rewards obtained after state s_t by its sub-behaviours. Note that the next state s_{t+1} in this case is the state encountered when the chosen action or behaviour terminates. The MaxQ hierarchy is represented by the tree in Figure 1. The different tasks it contains will now be described, specifically their possible choices, reward function, terminating states and the inputs and composition of their MLP. All hyper-parameters have been optimised using some preliminary experiments.

1) Root: The root sub-behaviour chooses the current sub-goal, which is either to open a door, reach the exit or fight with enemies. It decides this using the distance to various targets (doors, keys, exits), the amount of health it has and finally the safety of the current environment. This sub-behaviour terminates when the game is over, this occurs when the agent wins, the agent runs out of health (and dies) or the maximal amount of time steps is reached. Its reward function can be found in Table III. The MLP consists of an input layer consisting of 5 neurons, 1 hidden layer with 30 neurons and 3 output neurons.

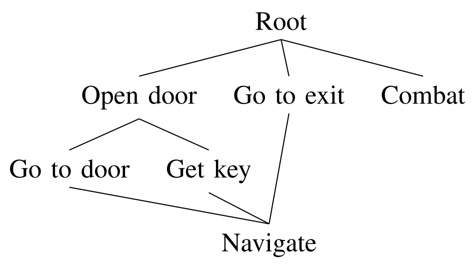


Fig. 1: MaxQ-Q AI hierarchy

TABLE III: Reward function for Root

Event	Type	Reward
Win the game	normal	10
Lose the game	normal	-10

2) Open door: The sub-behaviour 'go to door' either decides to pick up a key or go to a door. It has to decide this using the distances to the closest door and key. It also receives the amount of keys the player currently has. It terminates if it has opened a door or if there are no reachable doors. Its reward function can be found in Table IV. The MLP consists of an input layer consisting of 3 neurons, 1 hidden layer with 20 neurons and an output layer containing 2 outputs.

TABLE IV: Reward function for Open door

Event	Type	Reward
Opened door	normal	10
No reachable door	normal	-1

3) Navigate: The navigate sub-behaviour chooses a basic action, so either shoot or move in a direction. It gets the distance to the current target from the adjacent tiles, on which tiles enemies are present in a 7×7 square around the player and how many enemies it would hit if it shoots in a certain direction. It only terminates when it reaches its target. Its reward function can be found in Table V. The MLP consists of an input layer consisting of 63 neurons, 1 hidden layer with 300 neurons and 16 output neurons.

TABLE V: Reward function for Navigate

Event	Type	Reward
Target reached	pseudo	5
Moved away from target	pseudo	-5
Damaged by enemy	normal	-10
Killed by enemy	pseudo	-10

4) Combat: The combat sub-behaviour handles fighting with enemies, it can use the primitive actions move and shoot. As input it is given the current player health, on which tiles enemies are present in a 7×7 square around the player and how many enemies it would hit if it shoots in a certain direction. The combat sub-behaviour terminates if either an enemy unit has been killed or there are no enemies in any of its inputs. Its reward function can be found in Table VI. The MLP consists of an input layer consisting of 60 neurons, 1 hidden layer with 100 neurons and 16 outputs neurons.

TABLE VI: Reward function for Combat

Event	Type	Reward
Killed an enemy	normal	20
Damaged by enemy	normal	-10
Player died	pseudo	-10
No enemy in range	normal	-1

5) Get key, go to door, go to exit: The 'get key', 'go to door' and 'go to exit' sub-behaviours are different from the other sub-behaviours in that they can only choose the navigate sub-behaviour as an action. They are in the hierarchy only to give the navigate sub-behaviour the correct target (by using the inputs representing the relative distances to the target for each adjacent tile) and to keep track of rewards. All of the sub-behaviours terminate when their target has been reached (key, door or exit) or when the target is not reachable. The reward functions for get key, go to door and go to exit can be found in Tables VII, VIII and IX respectively.

TABLE VII: Reward function for get key

Event	Type	Reward
Picked up the key	normal	5
Key not reachable	normal	-1

TABLE VIII: Reward function for go to door

Event	Type	Reward
Opened door	normal	0
Reached door without key	normal	-1
Door not reachable	normal	-1

TABLE IX: Reward function for go to exit

Event	Type	Reward
Reached exit	normal	0
Door not reachable	normal	-1

B. Dynamic Hierarchical Reinforcement Learning

Our novel hierarchical AI consists of a root behaviour which can choose from various sub-behaviours. Each (sub-)behaviour has its own reward function which is not dependent on the reward of other behaviours. Behaviours can either execute primitive actions or delegate the decision to another sub-behaviour until a primitive action has been determined as seen in Algorithm 1. After the action has been performed all sub-behaviours that have chosen an action or sub-behaviour are trained using the Q-learning update rule with the starting state, the reached state and the rewards obtrained. Each sub-behaviour for which the following two conditions hold are also trained: the sub-behaviour can call the primitive action performed by the agent and the starting state is a state in which

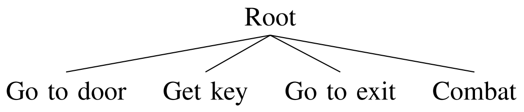


Fig. 2: dHRL AI hierarchy

the goal of the sub-behaviour is reachable. A counter example for the second situation would be when there are no enemies around the player or in shooting range, so that the combat behaviour would not learn since its goal (killing enemies) is not possible. Algorithm 2 shows how multiple behaviours can be updated at the same time on an experience.

Algorithm 1 Choosing action

```
action = Root
while action is not a primitive action do
    behaviour = action
    action = behaviour.getAction()
end while
```

Algorithm 2 Training behaviours

```
$a_t$ = chosen action
$s_t$ = state in which action was chosen
$s_{t+1}$ = next state
$r_t$ = list of rewards received before reaching the next state
for b in list of behaviours do
    if b.goal_is_reachable($s_t$) and b.choosable($a_t$) then
        b.update($s_t$, $a_t$, $r_t$, $s_{t+1}$)
    end if
end for
```

The main advantage of this system compared to MaxQ is that the agent does not get stuck in specific sub-tasks. Instead the agent can dynamically react to changes in its environment. Because of this, sub-behaviours can be made to be even more specialised on a single sub-goal. For example, the navigation behaviour does not need to learn how to evade enemies because the root-behaviour could simply choose the combat behaviour when enemies get close by.

The structure of the hierarchy is represented by the tree in Figure 2. It should be noted however that although go to door, get key and go to exit are displayed as different leaves, they actually share the same neural network. This underlying neural network is trained to give the best action based on the distance to the target from the 4 tiles directly adjacent to the agent.

1) Root: The root sub-behaviour chooses the current sub-goal, which is either to go to a door, get a key, go to an exit or fight with an enemy. It gets positively rewarded when a sub-goal is finished successfully, but choosing an impossible task, getting damaged by an enemy or losing the game is rewarded negatively. The exact rewards can be found in Table X. For each sub-goal it is easy to check whether it is currently possible to reach it. For the three navigation goals the AI simply checks whether the target is reachable, while the combat behaviour is deemed impossible if there are no enemies in the current state representation, or in other words: the safety input equals 1. The neural network consists of 6 input neurons, 1 hidden layer with 50 neurons and 4 output neurons.

TABLE X: Reward function for Root

Event	Reward
Win the game	10
Got a key	5
Opened a door	10
Sub-goal not possible	-1
Killed an enemy	20
Damaged by an enemy	-10
Lose the game	-10

2) Combat: As the name suggests the combat sub-behaviour handles combat related actions. To this end it gets rewarded for killing enemy units, but getting damaged, walking into walls and missing an enemy while shooting are punished. The exact rewards can be found in Table XI. As inputs it uses the current player health, on which tiles enemies are present in a 7×7 square around the player and how many enemies it would hit given it shoots in a direction. The neural network consists of 60 input neurons, 1 hidden layer with 100 neurons and 16 outputs neurons.

TABLE XI: Reward function for Combat

Event	Reward
Killed an enemy	5
Killed by an enemy	-10
Damaged by an enemy	-2
Walked into a wall	-1
Missed a shot	-2

3) Get key, Go to door and Go to exit: The Get key, Go to door and Go to exit sub-behaviours are all extremely similar and share the same underlying neural network. These behaviours get rewarded when their specific target has been reached, but if the agent moved away from the target or the behaviour walks against a wall, the agent is punished. The exact rewards can be found in Table XII. As input the behaviours only get the relative distance to their target from the four tiles adjacent to the agent. The neural network consists of 4 input neurons, 1 hidden layer with 50 neurons and 16 outputs neurons.

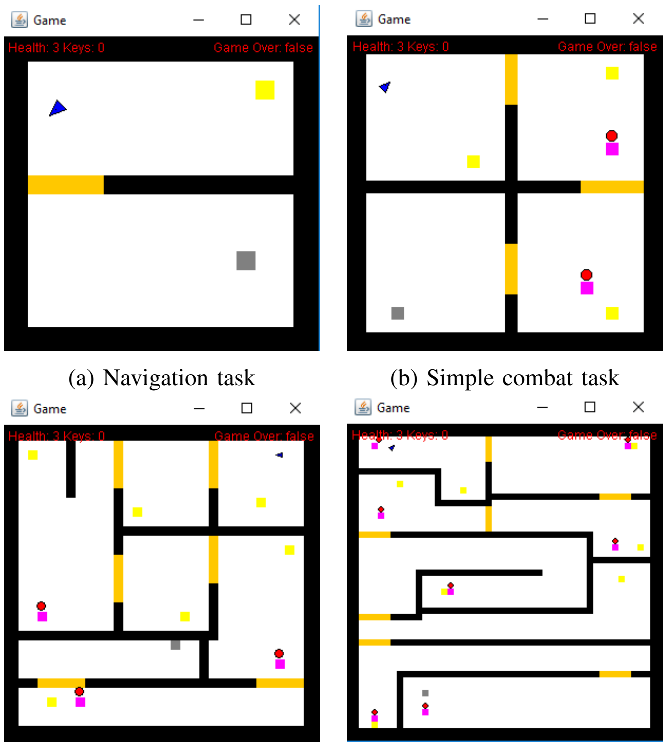
IV. Experiments and Results

A. Testing Setup

The two AIs are tested in 3 environments scaling up in difficulty. Each AI is trained and tested using 200 simulations with

TABLE XII: Reward function for the navigation behaviours

Event	Reward
Reached target	5
Walked away from target	-5
Walked into a wall	-1



(a) Navigation task (b) Simple combat task

(c) Hard combat task (d) Training map

Fig. 3: The different maps on which the AIs are tested. Black tiles represent walls. White tiles represent floors. Yellow tiles represent keys. Orange tiles represent locked doors. Purple tiles represent spawners. Red circles represent enemy ghosts. Grey tiles represent the exit.

a length of 500 games (epochs) in each of the environments. The environments are given in Figure 3. The first environment is a simple navigation map as can be seen in Figure 3a, it has a size of 16×16 tiles and consists of 2 rooms, in which the AIs only have to learn to first pickup the key, then open the door and finally reach the exit. The second map, found in Figure 3b, is larger with a size of 24×24 and consists of four rooms, two of which contain a spawner that generates enemy units. The AIs will have to pick up keys and open doors, and also have to learn how to deal with enemy units. Evasive manoeuvres are enough to reach the end of the level but killing the enemy units results in a higher score. The final map, which is shown in Figure 3c, is a large map with a size of 32×32 and consists of seven rooms of different sizes. The level has been setup in such a way that switching between navigation and combat on the fly is necessary because evading the enemy units is not possible in all locations.

The dHRL AI will also be tested on how well it generalises what it learned on one map. To that end the AI will be trained on the map found in Figure 3d which has a size of 48×48 and tested on the hard combat task without further training. The training map has been designed so that it contains most if not all possible situations a player would encounter on an arbitrary map. So the AI should be able to apply what it learned in these

TABLE XIII: Performance of both AIs on various tasks after 500 games of training. Results are averaged over 200 simulations.

Task	MaxQ-Q AI			dHRL AI		
	Score		Win rate	Score		Win rate
	Mean	Standard error		Mean	Standard error	
Navigation	109.7	2.1	0.97	114.9	3.4×10^{-3}	1.0
Simple combat	86.2	4.5	0.52	160.7	1.7	0.97
Hard combat	52.5	2.1	0.02	216.3	2.1	0.96

situations in other environments.

During training and testing the MaxQ-Q and dHRL AIs share similar settings which were found to perform best. During training both start with a temperature of 4 for the Boltzmann exploration algorithm which is multiplied by 0.98 after each game, to a minimum of 0.1. Their neural network learning rates are initialised slightly differently, for MaxQ-Q it starts at 0.001 while for dHRL it starts at 0.0005. Both have a decay after each game of 0.995.

We found that when a greedy policy is used during testing, both AIs sometimes got stuck in small loops of states. When the Boltzmann exploration algorithm is used with a low temperature, it significantly improves performance for both systems. Hence both use the Boltzmann algorithm with a temperature of 0.1 during performance tests.

B. Results

As can be seen in Figures 3.2 to 3.4 the dHRL AI outperforms the MaxQ-Q AI on all tasks. For the simple navigation task its mean score reaches the maximum score after about 200 epochs and although MaxQ-Q comes close, it does not reach the same performance. Table XIII also shows that the dHRL AI obtains a win rate of 1.0, while the MaxQ-Q AI only reaches a final win rate of 0.97.

In Figure 3.3 it can be observed that when combat is introduced the difference in scores of the two AIs grows. The dHRL AI learns faster than the MaxQ-Q AI and has a higher average score when it stabilises after around 200 epochs. Table XIII also shows that the difference between win rates also grows significantly. The dHRL reaches a win rate of 0.97 while the MaxQ-Q AI only reaches 0.52.

Figure 3.4 shows that the dHRL AI still performs well on the hard combat task and its mean score even increases compared to the simple combat task. This is because there are more enemy units that can be killed. The win rate is only lowered by 0.01 which indicates that the AI could likely learn to finish even harder and longer levels. The MaxQ-Q AI however scored less points on the hard combat task and its win rate dropped to 0.02.

The performance of dHRL drops however when it needs to generalise what it learned to different maps. As shown in Figure 3.5, its average score drops about 70 points compared to when it was trained on the same map. Its win rate also dropped from 0.96 to 0.59. Please have a look at: https://youtu.be/LJpuK3eWQeQ for a video showing the behaviour of agents trained with the 2 systems.

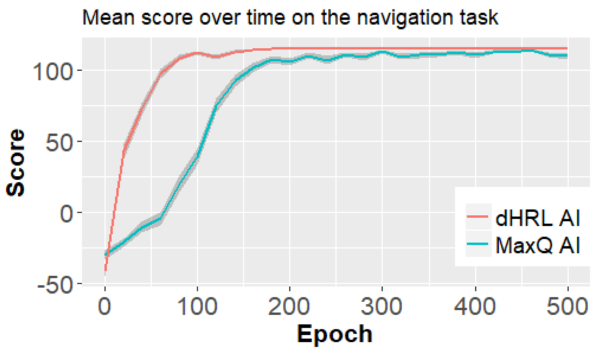


Fig. 3.2: Graph that shows average score and standard error for both AIs over time in the navigation task

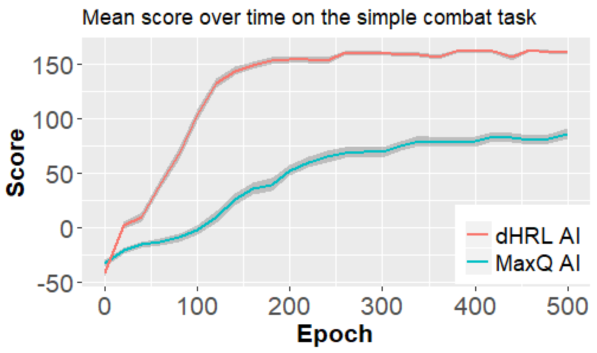


Fig. 3.3: Graph that shows average score and standard error for both AIs over time in the simple combat task

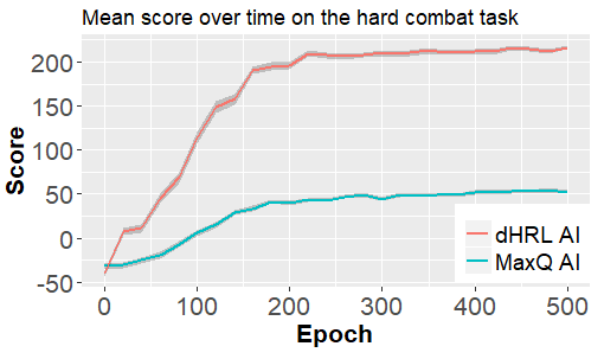


Fig. 3.4: Graph that shows average score and standard error for both AIs over time in the hard combat task

C. Discussion

The results clearly show that the dHRL algorithm significantly outperforms the MaxQ-Q AI on all of the tasks. On

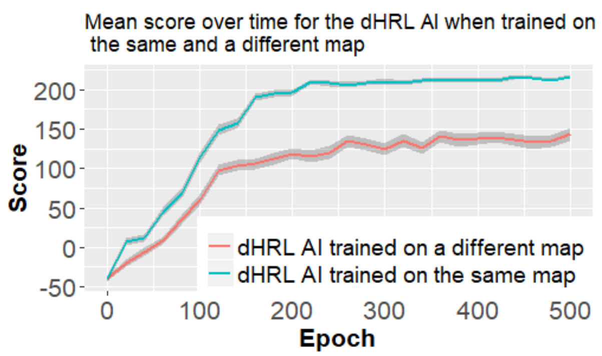


Fig. 3.5: Graph that shows average score and standard error for the dHRL when tested on the hard combat task while trained on the same map, compared to when it is trained on a different map.

the navigation task the results are still relatively close. This is because in the navigation task the optimal sequence of tasks is sequential. The agent should always first pick up a key, then open the door and finally go to the exit. There are no enemies hence dynamically switching between sub-behaviours does not give an inherent advantage. The difference in performance is most likely caused by how each AI learns. Since the dHRL AI uses multi-objective learning after each movement the navigation network is trained for each target (key, door, exit) that is currently reachable. When approaching the key, the door is also reachable and this speeds up learning significantly. The MaxQ-Q AI however is only trained on the current target, which makes learning take longer, even though the navigation behaviour is shared for different sub-tasks.

In the simple combat task the difference between the two AIs becomes bigger. This is because enemies are introduced and enemies create situations where it would be more optimal to switch to the combat behaviour while navigating and vice versa. The MaxQ-Q AI is not able to do this so its navigation behaviour now needs to learn how to either evade enemy units or attack them should they be encountered. In the dHRL AI the navigation behaviour only learns to navigate, because the root behaviour can simply choose to call the combat behaviour when enemies are close by. The reason why the MaxQ-Q AI still reaches a 0.52 win rate is because in the simple combat task walking around enemies is still a viable tactic, and the navigation behaviour is able to learn how to do that reliably in about 52% of the trials.

On the hard combat task the performance of dHRL is almost as good as on the simple combat task with a win rate of 0.96 versus 0.97 on the simple combat task, whereas the MaxQ-Q AI now reaches a win rate of 0.02. The main difference is the size of the map, and that there are now enemies which are almost impossible to dodge.

Finally the results show that although the dHRL AI can generalise what it learned on one map to other maps, it performs significantly worse. The problem is that the AI overfits on the map it trained on. This could be resolved by creating multiple maps on which the AI can train and then randomly choosing a map at the start of each epoch during training. That would minimise overfitting on specific maps.

V. Conclusions

We described a new way to create and train a hierarchical reinforcement learning system, where the programmer does not determine when behaviours start or finish. The combination of the hierarchical system, Q-learning, multi-layer perceptrons and higher-order inputs enabled the AI to win dungeon crawler levels of various difficulty within 500 training games. The system also has a limited ability to generalise the knowledge gained in a specific level to other levels.

In future work, we recommend to investigate if the ability to generalise can be improved given a better training setup: multiple levels to train, one of which is randomly chosen at the start of each epoch. The system should also be tested on how well it scales to more difficult tasks which need a hierarchy tree that is more than two layers deep.

References

[1] I. Szita, "Reinforcement learning in games," in *Reinforcement Learning State-of-the-Art*. Springer, 2012, pp. 539–577.
[2] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Springer, 2018, http://gameaibook.org.
[3] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning." in *AAAI*, vol. 16, 2016, pp. 2094–2100.
[4] G. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.
[5] M. Wiering and M. Van Otterlo, *Reinforcement Learning State-of-the-Art*. Springer, 2012, vol. 12.
[6] A. A. Markov, "The theory of algorithms," *Am. Math. Soc. Transl.*, vol. 15, pp. 1–14, 1960.
[7] A. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, pp. 341–379, 2003.
[8] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *J. Artif. Intell. Res.(JAIR)*, vol. 13, no. 1, pp. 227–303, 2000.
[9] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.
[10] S. J. Russell and A. Zimdars, "Q-decomposition for reinforcement learning agents," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 656–663.
[11] H. Van Seijen, M. Fatemi, J. Romoff, R. Laroche, T. Barnes, and J. Tsang, "Hybrid reward architecture for reinforcement learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 5392–5402.
[12] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.
[13] I. Ghory, "Reinforcment learning in board games." *Department of Computer Science, University of Bristol, Tech. Rep*, 2004.
[14] L. Bom, R. Henken, and M. Wiering, "Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs," in *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, 2013, pp. 156–163.
[15] R. Niel, J. Krebbers, M. M. Drugan, and M. A. Wiering, "Hierarchical reinforcement learning for real-time strategy games," in *Proceedings of the 10th International Conference on Agents and Artificial Intelligence*, vol. 2, 2018, pp. 470–477.
[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1.