Improving Efficiency of a Hierarchical Reinforcement Learning Algorithm

Bachelor's Project Thesis

Niclas Müller-Hof, s4351495, n.j.muller-hof@student.rug.nl,
Supervisors: R. (Rafael) Fernandes Cunha

Abstract: This thesis explores the efficiency of a hierarchical reinforcement learning algorithm using a fixed, state-dependent compression function to manage complex tasks with sparse rewards. To enhance learning efficiency, a multi-headed neural network architecture was proposed, enabling parameter sharing across subtasks while maintaining specialized outputs. However, experimental results indicated that this approach did not outperform the single neural network for each option, likely due to overgeneralization and insufficient capacity in shared layers. The study suggests future research should also focus on improving the multi-headed architecture to better balance shared and specialized components, potentially enhancing flexibility and overall performance.

1 Introduction

Reinforcement learning (RL) is a branch of machine learning in which an agent learns to make optimal decisions by interacting with its environment. RL has been successfully applied to a variety of domains, including robotics, game playing, and autonomous driving (Sivamayil et al. (2023)), where it enables systems to learn from experience and improve autonomously.

One significant challenge in reinforcement learning is the problem of sparse rewards. Sparse rewards occur when feedback from the environment is infrequent or delayed, making it difficult for the agent to learn the optimal policy. In such settings, the agent may struggle to discover which actions lead to rewards because the connection between actions and outcomes is obscured by the rarity of positive feedback. This often results in inefficient learning and poor performance, as the agent cannot effectively explore the state space or understand the consequences of its actions.

Hierarchical reinforcement learning (HRL) addresses the challenge of sparse rewards by decomposing complex tasks into simpler subtasks (Barto & Mahadevan (2003); Dietterich (1999)). HRL involves a hierarchical structure where high-level policies break down the problem into manageable subproblems, each with its own subgoals. This decomposition enables the agent to focus on achieving intermediate milestones, which are easier to learn and yield more frequent rewards. Specifically, HRL helps by creating a multi-level framework where higher-level policies guide the selection of lower-level actions, facilitating better exploration and learning in environments with sparse rewards.

HRL has garnered significant attention for its potential to efficiently solve complex tasks by decomposing them into manageable subtasks. Recent advancements have focused on improving the efficiency of HRL through various innovative approaches. For instance, an HRL method can exploit repeating sub-markov decision processes (sub-MDPs) to enhance statistical efficiency, providing a theoretical framework for model-based HRL algorithms (Wen et al. (2020)). Additionally, a proposed method using topologically sorted potential calculations to prioritize actions that lead to higher-level goal accomplishment, thereby enhancing learning efficiency (Zhou et al. (2023)). Furthermore, techniques that focus on subgoal discovery and reward shaping to balance exploration and exploitation to refine the performance of HRL systems (X. Gao et al. (2024); Wang et al. (2024)). These contributions collectively underscore the ongoing efforts to make HRL more efficient and ap-

plicable to complex, real-world tasks.

Building on these advancements, this thesis investigates the efficiency of an existing hierarchical reinforcement learning algorithm that uses a fixed, state-dependent compression function to break down complex tasks with sparse rewards into simpler, more manageable components. By doing so, the agent can define and focus on subtasks, which involve moving through different segments of the state space while simultaneously learning the policies for each extended action. The compression function facilitates the use of tabular methods at the higher levels of the hierarchy, enabling efficient exploration of extensive state spaces even when rewards are infrequent (Steccanella et al. (2020)). Each subtask architecture comprises two separate neural networks, one for the policy and one for the value function. To improve efficiency, I propose a multi-headed approach where a single neural network with multiple heads is used for each subtask (Lu et al. (2016); Neven et al. (2017)). This approach aims to enhance learning efficiency and performance compared to the single neural network for each policy, (Sener & Koltun (2018); Maninis et al. (2019)) by sharing network parameters across subtasks while maintaining specialized outputs for different policies and value functions (Lee et al. (2015); Kendall et al. (2017); Xu et al. (2018)).

Hence, I propose the research question: How can a neural network architecture be designed to integrate multiple option policies into a unified model within a hierarchical reinforcement learning framework, while effectively preserving optimal performance?

2 Background

To provide a comprehensive understanding of my implementation, we must first examine the pertinent background information and key theoretical frameworks.

In RL, the agent observes the current state, takes actions based on a policy, and transitions to a next state as a result of its actions. A policy is a strategy or a set of rules that defines the action the agent should take in each state to achieve the best possible outcome. The agent aims to learn this policy to maximize cumulative rewards over time. Through this iterative process of receiving feedback in the form of rewards and adjusting its actions accordingly, the agent continually learns from the consequences of its actions, allowing it to improve its performance and adapt to changing conditions. This approach is inspired by behavioral psychology and is well-suited for tasks where decision-making is sequential and outcomes are uncertain (Subramanian et al. (2022)).

2.1 Markov Decision Process

Consider a finite Markov Decision Process (MDP) (Puterman (1994, 2014)) which is a tuple $\mathcal{M} = \langle S, A, P, r \rangle$, where S is the finite state space, A is the finite action space, $P : S \times A = \Delta(S)$ is the transition kernel. Let $\Delta(S)$ be the probability simplex on S defined as $\Delta(S) = \{p \in \mathbb{R}^S : \sum_{s \in S} p(s) = 1, p(s) \geq 0 (\forall x)\}$. The function $r : S \times A \to \mathbb{R}$ is a reward function. The agent at time t observers a state $s_t \in S$, takes an action $a_t \in A$, obtains a reward r_t with expected value $\mathbb{E}[r_t] = r(s_t, a_t)$, and transitions to a new state $s_{t+1} \sim P(\cdot|s_t, a_t)$. We describe s_t, a_t, r_t, s_{t+1} as a *transition*.

Let π represent a stochastic policy $\pi : S \to \Delta(A)$, mapping states to probability distributions over actions. As said before the agent's objective is to learn a policy that maximizes the cumulative reward. To achieve this, we consider the discounted reward function. The expected reward of a policy π can be represented using a value function V^π, which is defined for each state $s \in S$ as follows:

$$V^\pi(s) = \mathbb{E}^\pi \left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \middle| s_1 = s \right]$$

here $\gamma \in (0, 1]$ is a discount factor used to account for the present value of future rewards, balancing immediate and long-term gains by weighting future rewards less heavily. The expectation is over the action $a_t \sim \pi(\cdot|s_t)$ and the next state $s_{t+1} \sim P(\cdot|s_t, a_t)$.

Another way to model the expected future reward is using an action-value function Q^π for state-action pair $(s, a) \in S \times A$ as follows:

$$Q^\pi(s, a) = \mathbb{E}^\pi \left[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t) \middle| s_1 = s, a_1 = a \right]$$

The objective of the agent is to learn a policy that maximizes the expected discounted cumulative reward $\eta(\pi)$. This is known as the optimal policy π^* which starts under some initial distribution $d_0 \in \Delta(S)$ over states as follows:

$$\eta(\pi) = E_{s\sim d_0}[V^\pi(s)]$$
$$\pi^* = \underset{\pi}{\text{argmax}}\eta(\pi)$$

2.2 Options

Given an MDP $\mathcal{M} = \langle S, A, P, r\rangle$, an option is a temporally extended action $o = \langle I^o, \pi^o, \beta^o\rangle$, where $I^o \subseteq S$ is an initiation set, $\pi^o : S \to \Delta(A)$ is a policy $\beta^o : S \to [0, 1]$ is a termination function (Sutton et al. (1999)). Adding options o to the action set A of $\mathcal{M}$ creates a Semi-Markov Decision Process (SMDP), which enables the agent to act on several timescales.

At the highest level also known as the manager in charge of the SMDP, the agent observers a state $s_t \in I^o$ at time t, select an option o_t, executes actions according to the option's policy π^o until it reaches the next state s_{t+k} in which the termination condition $\beta^o(s_{t+k})$ triggers. Even though the option takes multiple actions from the perspective of the manager in charge of the SMPD one action is taken.

In order, to train the policy π^o, it is common to define an option-specific reward function r^o, which defines an option-specific MDP $\mathcal{M}^o = \langle S, A, P, r^o\rangle$. The policy π^o is implicitly defined as the optimal solution to $\mathcal{M}^o$.

2.3 Task MDP Extension

A MDP, denoted as $\mathcal{M}$, can be extended to define a specific task by introducing additional states and actions. These additions enable an agent to interact with new objects within the environment. Formally, we define a task $\mathcal{T}$ using an extended MDP, $\mathcal{M}_\mathcal{T}$, which incorporates both the original and task-specific components. The expanded model is represented as $\mathcal{M}_\mathcal{T} = \langle S_i \times S_\mathcal{T}, A_i \cup A_\mathcal{T}, r_\mathcal{T}, P_i \cup P_\mathcal{T}\rangle$. Here, $S_\mathcal{T}$ includes task-specific states, and $A_\mathcal{T}$ includes additional actions that interact with these new states. The transition function $P_\mathcal{T}$ specifies the dynamics of these interactions, determining how the environment's state changes in response to the task-specific actions. The reward function $r_\mathcal{T}$ assigns values based on the outcomes of these interactions, promoting behaviors that are desirable within the task's context. The invariant states S_i, actions A_i, and transitions P_i remain the same across different tasks and are also shared between tasks, ensuring a consistent foundation upon which various tasks can be built (Steccanella et al. (2020)).

2.4 Invariant MDP and Regions

We introduce a concept where the agent accesses a partition of the invariant state space, denoted as $Z = \{Z_1, ..., Z_m\}$. This partition divides the state space S_i into distinct non-overlapping regions Z_i, where each region represents a subset of the entire state space. These regions are fundamental in constructing a Semi-Markov Decision Process (SMPD) over the invariant aspects of the state-action space, defined as $\mathcal{S} = \langle Z, O, P_Z\rangle$. Within this structure, Z represents the set of regions, O is a collection of options (complex actions composed of multiple primitive actions), and P_Z is the transition kernel that models the probabilities of moving from one region to another based on the selected options.

The neighborhood of a region z, denoted as $\mathcal{N}(z)$, comprises regions that can be directly reached from z through available actions. For each neighboring region z', we define an option $o_{z,z'}$, which is a policy specifically designed to navigate from region z to region z'. This option starts in region z and aims to terminate successfully in region z'. The definition of these options and their corresponding policies plays a crucial role in the efficient exploration and exploitation of the state space, facilitating the agent's learning and decision-making processes within the invariant framework of the MDP (Steccanella et al. (2020)).

2.5 Options MDP and Exploration Strategies

In the framework, we define a specialized MDP for each option, termed $\mathcal{M}_{z,z'} = \langle S_z, A_i, P_z, r_{z,z'}\rangle$. This MDP focuses on the states S_z within the initiating region z and its neighbors $\mathcal{N}(z)$. The action set A_i remains consistent with the invariant part of the MDP. This specialized MDP is crucial for evaluating and optimizing the options, as it directly

models the outcomes and rewards associated with transitioning from the region z to its neighbors.

The transition dynamics within this MDP, denoted as P_z, are carefully designed to reflect the combined probabilities of moving from any state in z to states in the neighboring regions. The reward function $r_{z,z'}$ is structured to incentivize reaching the target region z' directly by giving a positive reward for reaching the correct region and a negative reward for the wrong region, thereby promoting efficient navigation through the state space. This setup helps in refining the strategies associated with each option, enhancing the overall effectiveness of the decision-making process in the face of complex, multi-step tasks (Steccanella et al. (2020)).

2.6 Algorithm for Learning and Decision Making

The algorithm proposed here, named INVARIANTHRL (2.1), is designed to iteratively build and refine a model of the SMPD by exploring the environment through a sequence of options. Initially, the agent starts in a single region, identified by the state s and its corresponding region z, determined by a mapping function $f : S_i \to \mathbb{N}_+$. As the agent explores the environment, it dynamically constructs the set of regions Z and the associated options O, adjusting its strategy based on the observed outcomes and rewards.

Each exploration step involves selecting an option that is expected to yield the most information about the environment or the highest reward, depending on the specific objectives of the task at hand. The option execution, guided by the policy $\pi_{z,z'}$, provides empirical data that the agent uses to refine its understanding of the environment's dynamics.

2.7 Solving Task-Specific MDPs

As the agent continues to explore and expand its knowledge of the environment through the SMPD framework, it also needs to adapt these insights to solve specific task MDPs, $\mathcal{M}_{\mathcal{T}}$. Each task introduces unique states and actions, necessitating adjustments to the existing framework to accommodate these new elements. By integrating task-specific options into the decision-making process, the agent can effectively navigate both the invariant and task-specific elements of the environment, optimizing its strategies to maximize rewards and achieve task-specific goals. This integrated approach ensures that the agent can handle a wide range of scenarios and adapt its strategies to meet the challenges of diverse and dynamic environments (Steccanella et al. (2020)).

Algorithm 2.1 INVARIANTHRL

Input: Action set A_i, oracle compression function f
  $s \Leftarrow$ initial state, $z \Leftarrow f(s)$
  $Z \Leftarrow \{z\}, O \Leftarrow \{o^e_z\}$
  while within budget **do**
    $o \Leftarrow \text{GETOPTION}(z, O)$
    $s' \Leftarrow \text{RUNOPTION}(s, o, A_i), z' \Leftarrow f(s')$
    if $z' \notin Z$ **then**
      $Z \Leftarrow Z \cup \{z'\}$
      $O \Leftarrow O \cup \{o^e_{z'}\}$
    end if
    if $o_{z,z'} \notin O$ **then**
      $O \Leftarrow O \cup \{o_{z,z'}\}$
    end if
    $s \Leftarrow s', z \Leftarrow z'$
  end while

3 Contribution

In this part, I explain the implementation of the algorithm and the modifications made. We separate the hierarchical implementation into two parts, namely a manager and workers to effectively solve complex decision-making tasks. The manager is responsible for solving the task SMDP S_T, which involves high-level decision-making and coordination across various task states. The manager's primary role is to determine which option to execute in order to transition between abstract regions or states, guiding the overall strategy for task completion.

On the other hand, the workers are specialized agents responsible for solving the option Markov Decision Processes (MDPs) $M_{z,z'}$ or $M^z_{s,s'}$ for task-specific options. Each worker focuses on a particular option, learning the detailed policy and value functions necessary to transition between specific states or achieve particular sub-goals within the task framework. While the manager oversees the

broader task strategy, the workers handle the execution of these strategies by optimizing their respective option policies.

3.1 Manager

Given that the space of regions Z is small, the manager employs tabular Q-learning over the task SMDP $S_{\mathcal{T}}$. This process is detailed in Algorithm 3.1. Similar to Algorithm 2.1, the task state space $S_{\mathcal{T}}$ and option set $O_{\mathcal{T}}$ expand as the agent discovers new states and transitions. The manager maintains and updates Q-values for various tasks and options. For each task, it initializes a Q-table, which is dynamically updated as new regions and transitions are discovered.

To manage transitions effectively, the manager creates or retrieves invariant and task-specific options for transitioning between regions or tasks. If an option does not already exist for a given transition, a new worker is created to handle it. The manager updates its policy using the Q-learning update rule. It calculates the TD error based on the received reward and the maximum Q-value of the next state. This error is used to update the Q-value for the current state-option pair, allowing the manager to learn and improve its policies over time.

This implementation ensures that the manager effectively learns and updates the optimal policies for transitioning between regions. The manager becomes increasingly proficient at navigating the task by adapting to new discoveries and continuously improving through learning.

3.2 Worker

To facilitate the worker's ability to transition between abstract states z and z' (or task states s and s'), we employ two distinct neural networks: one for the policy and one for the value function. These networks are responsible for learning the correct behaviors through Self-Imitation Learning (SIL) and off-policy critic updates.

The original paper utilized a single neural network for each option. To improve the efficiency of this, I propose exploring a different approach.

In our approach, each network features a shared component and multiple heads specific to various options, allowing the worker to learn and update policies and value functions dynamically.

Algorithm 3.1 MANAGER

Input: Task action set $A_{\mathcal{T}}$, invariant SMDP $\mathcal{S}$
  $z \Leftarrow$ initial region, $s \Leftarrow$ initial task state
  $S_{\mathcal{T}} \Leftarrow \{s\}, O_{\mathcal{T}} \Leftarrow \emptyset$
  while within budget **do**
    $o \Leftarrow \text{GETOPTION}(\pi_{\mathcal{T}}, (z, s), O \cup O_{\mathcal{T}})$
    $(z', s), r \Leftarrow \text{RUNOPTION}((z, s), o, A_i \cup A_{\mathcal{T}})$
    $\text{UPDATEPOLICY}(\pi_{\mathcal{T}}, (z, s), o, r, (z', s'))$
    if $s' \notin S_{\mathcal{T}}$ **then**
      $S_{\mathcal{T}} \Leftarrow S_{\mathcal{T}} \cup \{s'\}$
    end if
    if $o_z^{s,s'} \notin O_{\mathcal{T}}$ **then**
      $O_{\mathcal{T}} \Leftarrow O_{\mathcal{T}} \cup \{o_z^{s,s'}\}$
    end if
    $(z, s) \Leftarrow (z', s')$
  end while

The Policy Network ($\pi_{\theta_{z,z'}}$), parameterized by θ, is tasked with learning the policy that dictates the actions a the agent should take to transition between states s. The network includes shared layers that process the input state s into a shared representation. This shared representation is then fed into option-specific heads, each consisting of a fully connected layer with 64 units followed by a Softmax activation function. This setup allows the network to output a probability distribution over actions for the selected option. The architecture is flexible, enabling the addition of new heads as new options are discovered. On the other hand, the Value Network ($V_{\psi_{z,z'}}$), parameterized by ψ, estimates the value function for each option. It also includes shared layers that process the input state s into a shared representation, which is then passed through option-specific heads. Each head ends with a linear layer that outputs the value estimation for the corresponding option. Like the policy network, the value network can dynamically add new heads for new options, ensuring it can handle an expanding set of tasks.

The agent uses mini-batch stochastic gradient descent to minimize a composite loss function, which includes the policy loss, the entropy regularization term, and the value loss. The overall loss function is expressed as

$$L(\theta, \psi) = L(\hat{\eta_\theta}) + \alpha H_\pi + L(\hat{V_\psi})$$

Here, $L(\hat{\eta_\theta})$ represents the loss associated with the policy network, encouraging the agent to imitate

successful past experiences through self-imitation. The term αH_π is the entropy regularization component, ensuring sufficient exploration by penalizing low-entropy (overly deterministic) policies. Finally, $L(\hat{V_\psi})$ is the loss associated with the value network, measured as the mean squared error between the predicted values and the target values.

The training process involves on-policy training, where the agent collects transitions through interactions with the environment and uses these samples to update both the policy and value networks. Additionally, off-policy critic updates with relabeling are employed to accelerate the learning of correct behaviors. This approach, similar to Hindsight Experience Replay (HER) (Andrychowicz et al. (2017)), involves relabeling failed transitions to interpret unsuccessful experiences in the context of achieving alternative goals. In addition, the worker implements Self-Imitation Learning (SIL). This method involves storing experiences in a replay buffer and periodically sampling from this buffer to reinforce successful behaviors. Transitions are stored in the memory buffer, and batches are sampled from this buffer to update the networks based on these samples. SIL provides an exploration bonus by encouraging the worker to repeat actions that previously led to high rewards, enhancing the learning process.

The key difference between the modified worker and the previous worker lies in their network architectures and how they manage multiple options. The previous worker employs separate, static neural networks for each option, with each network independently learning the policy and value functions for its specific task. In contrast, the modified worker uses shared layers in both the policy and value networks, which process the input state into a common representation. This shared representation is then fed into option-specific heads that are dynamically added as new options are discovered. The difference between the two architectures can be seen in Figure 3.1. This approach allows the modified worker to efficiently scale and adapt to new tasks without requiring a complete reconfiguration of the network architecture for each new option.

This framework ensures that the agent effectively learns the policies and value functions required to transition between states, leveraging both on-policy and off-policy learning techniques to optimize performance.

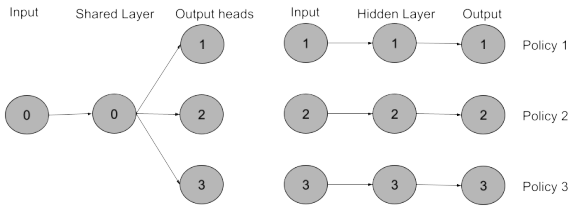

Figure 3.1: Neural Network representation of the worker's architecture. Showing the neural network with multiple heads for each option (on the left) and the single neural network for each option (on the right)

3.3 Properties

The multi-headed architecture of the workers, exemplified by the multi-headed policy network, offers several theoretical key advantages over creating multiple separate policy networks for each option. One of the primary potential benefits is efficiency (Huang (2024); Vandenhende et al. (2019)). The shared layers in the multi-headed policy network allow for common computations to be done once per input state, reducing redundant calculations. This would lead to significant savings in computational resources and time (Kokkinos (2016)). Moreover, the overall memory usage would be lower since the shared layers are not duplicated across multiple networks (Lu et al. (2016); Guo et al. (2018); Kendall et al. (2017)). Only the heads, which are typically smaller, require additional memory for each new option, which would result in a more efficient memory footprint.

For example, consider an environment with a length and width of 8 creating a total of 64 states s and these states split into 4 different regions z. Giving each region 16 states s. Using this we can calculate the total parameter count using the following formula:

$$P = (s \times h) + (h \times a) + h + a$$

Where h represents the number of neurons in the hidden layer and a represents the number of neurons in the output layer. In a shared neural network with multiple heads for each option, the shared layer has $16 \times 64 + 64 = 1088$ parameters. Each head, contributes $4 \times (64 \times 4 + 4) = 1040$ parameters. Thus, the total number of parameters

is $1088 + 1040 = 2128$. In contrast, using separate neural networks for each option results in $4 \times ((16 \times 64) + (64 \times 4) + 64 + 4) = 5392$ parameters for the same four options. This setup leads to a parameter reduction of approximately 60.5%, calculated as

$$\left(1 - \frac{2128}{5392}\right) \times 100 \approx 60.5\%$$

Furthermore, the multi-task learning capability of the network would enable the agent to learn and generalize across multiple tasks. By sharing common representations in the shared layers and maintaining option-specific and task-specific outputs in the heads, the network can effectively handle a variety of tasks simultaneously (Lu et al. (2016); Kendall et al. (2017)).

The multi-headed architecture also promotes improved generalization (Vandenhende et al. (2019)). Shared representations learned in the common layers would allow the network to leverage information across different options, which would lead to better performance on new or unseen tasks. Additionally, the shared layers act as a form of regularization, encouraging the network to learn more robust and generalized features that are effective across multiple heads. (Y. Gao et al. (2019)) This regularization effect should help prevent overfitting and enhance the network's ability to generalize well across different tasks (Ghosh et al. (2018)).

Furthermore, this architecture could be expected to learn faster and take less time compared to implementing separate neural networks for each option. By reusing shared computations, the training process becomes more efficient, reducing the overall training time. (Lu et al. (2016); Neven et al. (2017)) The shared layers enable the network to benefit from transfer learning, where knowledge gained from one task can aid in learning another, thereby speeding up convergence (Huang (2024); Vandenhende et al. (2019)).

Overall, the policy network's multi-headed architecture combines efficiency, flexibility, adaptability, and improved generalization, making it in theory a powerful and versatile solution for complex, multi-task environments. This architecture should ensure that the worker is able to efficiently learn and update the optimal policies and value functions for each option, dynamically adapting to new discoveries.

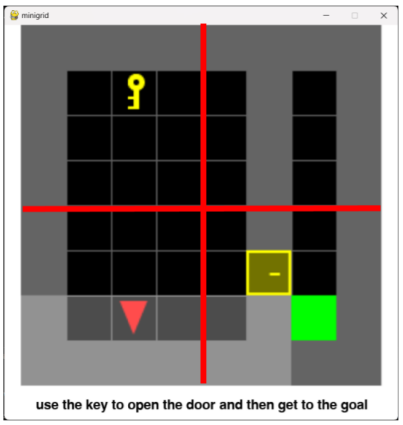

Figure 4.1: Key-Door-Treasure GridWorld 8x8 environment with red lines showing the compression function splitting the environment into 4 regions

4 Experiments

To compare this work with the study detailed in the paper "Hierarchical Reinforcement Learning for

Table 4.1: Hyperparameters used for the experiments

Hyperparameter	value
Network Architecture	
Shared-Layer	FC(64)
Heads	FC(64)
Worker	
Learning rate	0.0007
Gamma (Discount Factor)	0.99
SIL Replay Buffer size	10^4
SIL Batch size	512
SIL loss weight	1
SIL value loss weight	0.01
Epsilon (numerical stability)	1e-8
SIL updates per iteration	4
Manager	
Steps per iteration update	6
Bias correction	0.1
Gamma (Discount Factor)	0.9
Epsilon Decay	0.95
Environment	
Key-door-Treasure	8x8, 16x16
Number of iterations	1000

Efficient Exploration and Transfer." (Steccanella et al. (2020)) I employed several key evaluation metrics: total rewards per episode, total steps taken per episode, and the overall time taken to run the algorithm. Total rewards and steps were used to assess the performance of the models, while the total time taken provided a measure of their computational efficiency.

To perform these evaluations I use two different sizes of a Minigrid environment namely the Key-Door-Treasure GridWorld 8x8 and 16x16. In both of these environments, the agent has to pick up a key to open a door, enabling the agent to reach the goal. The invariant part of the state consists of the agent's location, and the compression function f imposing a grid-like structure on top of the environment as shown in figure 4.1. Important to note is that the compression function for the 8x8 Key-Door-Treasure GridWorld splits the environment into 4 regions, where each region has a total of 16 states. Whereas the 16x16 environment is split into 16 regions resulting in also 16 states per region.

For both environments, the agent receives a reward for each intermediate goal. Meaning a reward is given for picking up the key, opening up the door, and reaching the goal. Another important property to note is that the agent takes a random action with a probability of 20 %. The agent also has a budget of 300 time steps before the environment resets. I average the results over 5 seeds and the experiment is run for 1000 episodes. In table 4.1 the hyperparameters are shown to run the algorithm. In all 5 of the runs, the starting location of the agent, key, and door is random but the location of the goal is always in the bottom right corner of the environment.

To validate my hypothesis, I expect the algorithm to learn and converge to achieve the maximum sum of rewards in the environment consistently. Additionally, I anticipate that the algorithm will converge to the minimum number of steps required to solve the environment. Crucially, for my hypothesis to be accepted, the total time taken after 1000 episodes should be less for the multi-headed neural network compared to the single neural network for each option.

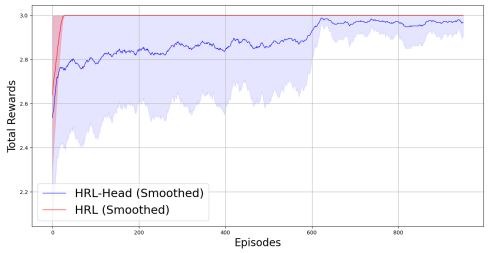

Figure 4.2: Total rewards for 8x8 environment

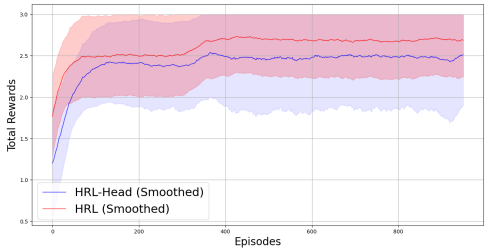

Figure 4.3: Total rewards for 16x16 environment

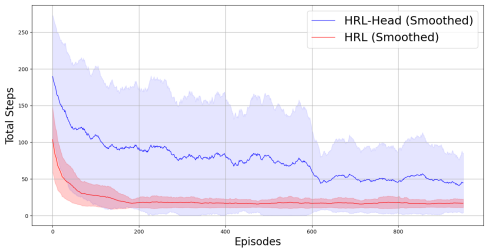

Figure 4.4: Total steps for 8x8 environment

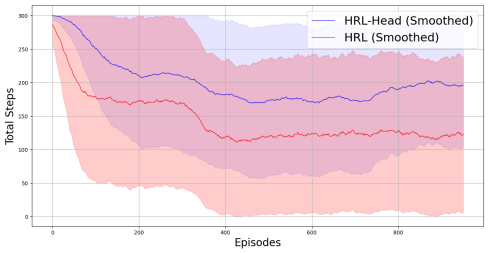

Figure 4.5: Total steps for 16x16 environment

4.1 Results

In the figures, my algorithm is labeled as HRL-Head, while the implementation of the paper "Hierarchical Reinforcement Learning for Efficient Exploration and Transfer" is labeled as HRL (Steccanella et al. (2020)). The results indicate that the multi-headed implementation is indeed still learning. In figure 4.2 by blue it is evident that the total rewards show a steady increase. After approximately 1000 episodes, the rewards stabilize at just below 3, though they have not fully converged to the maximum reward and still exhibit a high amount of variation. In contrast, the single neural network for each option shown by the red line in figure 4.2 converges more quickly, maintaining a steady total reward of 3 for the remainder of the episodes.

In Figure 4.4 a similar pattern is observed in the total steps taken per episode: the multi-headed implementation shows a clear decrease in steps but with significant variation, failing to converge to the optimal number of steps required to solve the environment. Conversely, the single neural network implementation demonstrates a rapid decrease in steps and quickly converges to the minimum number needed to solve the environment. In the 16x16 environment, Figure 4.3 and 4.5 shows that this trend persists, with the single neural network for each option achieving higher average rewards and requiring fewer steps than the multi-headed neural network.

In Table 4.2 it can be seen that the single neural network for each option takes less time for 1000 episodes than the multi-headed neural network.

Table 4.2: Average time take in hours for the multi-headed neural network and the single neural network for each option in the 16x16 and 8x8 key-door-treasure environment for 1000 episodes

	8x8	16x16
Multi-head	10.71	47.88
Single	9.14	46.10

5 Discussion

Based on the results, we cannot accept our hypothesis, as the time taken has actually increased when using the multi-headed implementation compared to a single neural network for each option. The overall increase in time is likely due to the algorithm's inability to learn the optimal policy π_o for each region. This inefficiency results in the algorithm taking more steps per episode, thereby increasing the time required for each episode. The multi-headed approach has led to worse performance due to several potential reasons.

One significant issue is overgeneralization. In the multi-headed neural network, shared layers are designed to learn common features across multiple policies. However, these shared features might be too general and not specialized enough for the specific requirements of each individual policy. This lack of specialization can lead to suboptimal performance, as the shared layers may fail to capture the nuanced details necessary for each policy to operate optimally. To address this, it is essential to design and tune the network carefully to ensure that while the shared layers capture essential common features, they do not lose critical policy-specific details.

Another factor to consider is insufficient capacity in the shared layers. The shared layers might not have the capacity to adequately learn and represent the complexities inherent in all the policies they serve. This limitation can result in underfitting, where the network fails to capture important features for each policy, thereby compromising performance. To mitigate this, the capacity of the shared layers can be increased by adding more neurons or layers.

Furthermore, inadequate head networks can contribute to the suboptimal performance of the multi-headed network (Narayanan et al. (2021)). If the head networks are too simple, they may not effectively leverage the features learned by the shared layers. This inadequacy can lead to poor policy outputs and decreased overall performance. Ensuring that the head networks have sufficient capacity and complexity is vital. A higher capacity head network can potentially process the shared features more effectively and produce high-quality policy outputs, thereby enhancing the entire network's performance.

Another limitation regarding the whole algorithm is that the compression function is hard-coded in our approach to create specific regions, enabling the agent to break down complex tasks into simpler, more manageable components. However, it is feasible to design an automatic compression function, which could enhance the flexibility and adaptability of the algorithm. Implementing an automatic compression function would involve using clustering algorithms or neural network-based methods to dynamically identify and segment regions based on state-space similarities and task-specific features. For instance, techniques like K-means clustering (MacQueen (1967)). By Leveraging this, the algorithm could automatically adapt to different environments and tasks, potentially improving its overall performance and efficiency.

6 Conclusion

This paper explored the implementation and efficiency of a hierarchical reinforcement learning algorithm using a fixed, state-dependent compression function to decompose complex tasks with sparse rewards into simpler subtasks. The introduction of a multi-headed neural network architecture aimed to enhance learning efficiency by sharing network parameters across subtasks while maintaining specialized outputs. However, experimental results indicated that the multi-headed approach did not outperform the single neural network for each option, potentially due to overgeneralization, insufficient capacity in shared layers, and inadequate head networks.

References

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., ... Zaremba, W. (2017). Hindsight experience replay. *CoRR*, *abs/1707.01495*.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*, 41-77.

Dietterich, T. G. (1999). Hierarchical reinforcement learning with the MAXQ value function decomposition. *CoRR*, *cs.LG/9905014*.

Gao, X., Liu, J., Wan, B., & An, L. (2024, 05). Hierarchical reinforcement learning from demonstration via reachability-based reward shaping. *Neural Processing Letters*, *56*. doi: 10.1007/s11063-024-11632-x

Gao, Y., Ma, J., Zhao, M., Liu, W., & Yuille, A. L. (2019, June). Nddr-cnn: Layerwise feature fusing in multi-task cnns by neural discriminative dimensionality reduction. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition (cvpr).*

Ghosh, S., Mercier, A., Pichapati, D., Jha, S., Yegneswaran, V., & Lincoln, P. (2018). *Trusted neural networks for safety-constrained autonomous control.*

Guo, M., Haque, A., Huang, D.-A., Yeung, S., & Fei-Fei, L. (2018, September). Dynamic task prioritization for multitask learning. In *Proceedings of the european conference on computer vision (eccv).*

Huang, G. (2024, 03). Dynamic neural networks: advantages and challenges. *National Science Review*, nwae088. doi: 10.1093/nsr/nwae088

Kendall, A., Gal, Y., & Cipolla, R. (2017). Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. *CoRR*, *abs/1705.07115*.

Kokkinos, I. (2016). Ubernet: Training a 'universal' convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. *CoRR*, *abs/1609.02132*.

Lee, S., Purushwalkam, S., Cogswell, M., Crandall, D. J., & Batra, D. (2015). Why M heads are better than one: Training a diverse ensemble of deep networks. *CoRR*, *abs/1511.06314*.

Lu, Y., Kumar, A., Zhai, S., Cheng, Y., Javidi, T., & Feris, R. S. (2016). Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. *CoRR*, *abs/1611.05377*.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations..

Maninis, K., Radosavovic, I., & Kokkinos, I. (2019). Attentive single-tasking of multiple tasks. *CoRR*, *abs/1904.08918*.

Narayanan, A. R., Zela, A., Saikia, T., Brox, T., & Hutter, F. (2021). Multi-headed neural ensemble search. *CoRR*, *abs/2107.04369*.

Neven, D., Brabandere, B. D., Georgoulis, S., Proesmans, M., & Gool, L. V. (2017). Fast scene understanding for autonomous driving. *CoRR*, *abs/1708.02550*.

Puterman, M. L. (1994). Markov decision processes: Discrete stochastic dynamic programming..

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons.

Sener, O., & Koltun, V. (2018). Multi-task learning as multi-objective optimization. *CoRR*, *abs/1810.04650*.

Sivamayil, K., Rajasekar, E., Aljafari, B., Nikolovski, S., Vairavasundaram, S., & Vairavasundaram, I. (2023). A systematic study on reinforcement learning based applications. *Energies*, *16*(3). doi: 10.3390/en16031512

Steccanella, L., Totaro, S., Allonsius, D., & Jonsson, A. (2020). Hierarchical reinforcement learning for efficient exploration and transfer. *CoRR*, *abs/2011.06335*.

Subramanian, A., Chitlangia, S., & Baths, V. (2022). Reinforcement learning and its connections with neuroscience and psychology. *Neural Networks*, *145*, 271-287. doi: https://doi.org/10.1016/j.neunet.2021.10.003

Sutton, R. S., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1), 181-211. doi: https://doi.org/10.1016/S0004-3702(99)00052-1

Vandenhende, S., Brabandere, B. D., & Gool, L. V. (2019). Branched multi-task networks: Deciding what layers to share. *CoRR*, *abs/1904.02920*.

Wang, K., Ruan, J., Zhang, Q., & Xing, D. (2024). Efficient hierarchical reinforcement learning via mutual information constrained subgoal discovery. In B. Luo, L. Cheng, Z.-G. Wu, H. Li, & C. Li (Eds.), *Neural information processing* (pp. 76–87). Singapore: Springer Nature Singapore.

Wen, Z., Precup, D., Ibrahimi, M., Barreto, A., Van Roy, B., & Singh, S. (2020). On efficiency in hierarchical reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (Vol. 33, pp. 6708–6718). Curran Associates, Inc.

Xu, D., Ouyang, W., Wang, X., & Sebe, N. (2018). Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing. *CoRR*, *abs/1805.04409*.

Zhou, Z., Shang, J., & Li, Y. (2023). Enhancing efficiency in hierarchical reinforcement learning through topological-sorted potential calculation. *Electronics*, *12*(17). doi: 10.3390/electronics12173700

7 Appendix

The code for the single neural network implementation, as well as the multi-headed implementation, can be found on this GitHub page: `https://github.com/Niclas-J-M/BP`

8 Acknowledgements

I am profoundly grateful to Lecturer Rafael Fernandes Cunha for his invaluable guidance and support throughout my bachelor project. His expertise and insightful feedback have been instrumental in shaping my work. I deeply appreciate his dedication to mentoring and the significant time he invested in my academic development.