university of groningen

faculty of science and engineering

Department of Artificial Intelligence
Master's thesis

Hierarchical Reinforcement Learning Approach to Lifelong Learning

Author
Diego Cabo Golvano
S-3736555

First Supervisor
Dr. Marco Wiering
Artificial Intelligence
University of Groningen

Second Supervisor
Dr. Hamidreza Kasaei
Artificial Intelligence
University of Groningen

November 30, 2020

Abstract

Autonomous systems such as the robots sent on exploratory missions in space or underwater must have the ability to learn new things and recognize new objects after they have been deployed, and must take decisions during extended periods of time within a constantly changing environment. Lifelong machine learning is a key component for those systems and something we need in order to achieve real artificial general intelligence. By levering the power of hierarchical reinforcement learning we can create systems that can effectively and efficiently retain and reuse knowledge, a key aspect of lifelong machine learning. In this thesis we design a system that is able to learn new tasks throughout its lifetime, while still being able to remember how to solve tasks that it had already learnt. We create a simulated environment in which to test this new system and compare it to a more standard Q-learning approach. The empirical results confirm that the proposed system is able to function in the simulated environment and that, with more research involved, it could become a viable solution for the aforementioned real life systems.

Keywords — Hierarchical Reinforcement Learning; Lifelong Machine Learning; Autonomous Systems; Machine Learning

Acknowledgements

I would like to express my gratitude to my thesis supervisors Dr Marco Wiering and Dr. Hamidreza Kasaei. They always provided me with valuable insights on this project and it would not have been possible to finish it without them.

I would also like to thank the University of Groningen and its Center for Information Technology for allowing me to use their infrastructure during my courses as well as the access to the Peregrine cluster for my thesis. All the computational resources needed for this thesis were vital for the successful termination of it.

I would also like to thank my parents for always giving me their support and unconditional love. This journey would not have been possible without them.

Lastly, I want to thank my wife for accompanying me in this journey. I know that it has not been an easy path and I'll always be grateful for having had her by my side during this experience.

Nomenclature

AGI	Artificial General Intelligence
AI	Artificial Intelligence
ER	Experience Replay
HRL	Hierarchical Reinforcement Learning
LML	Lifelong Machine Learning
ML	Machine Learning
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
RL	Reinforcement Learning

Contents

Chapter 1

Introduction

Artificial intelligence might be one of the most popular topics nowadays. Vast amounts of resources are being poured into universities and big research labs. Every company that wants to survive the new decade is trying to incorporate it into their product line and services. Simply stating that some product uses AI technology sells more. All this hype around AI has originated cases such as companies claiming to incorporate AI into their systems when all they do are simple operations.

But, what is intelligence? If there was a simple and intuitive definition we wouldn't have cases as the aforementioned. People would be able to judge easily if something had AI or not. However, it's not easy to come up with a single consensus definition of intelligence. Even among experts, there are several different definitions of that concept. One of the fathers of AI, John McCarthy, defined it as: *"Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines"* (McCarthy, 1998). The definition of intelligence is very much linked to our understanding of human intelligence, or equivalently, the human brain. Thus, since our understanding of the brain is limited, so is our capacity to define the boundaries of what is and what isn't intelligent. The term Artificial General Intelligence (AGI) was coined to refer to the hypothetical intelligence of a machine that has the capacity to understand or learn any intellectual task that a human being can.

One characteristic that we can observe in organisms with high degrees of intelligence, is their ability to learn continuously throughout their lives. Humans are exceptionally efficient at storing new skills and reusing them to solve new problems. If a person learns how to ride a bike, s/he can then learn how to ride a motorbike much faster, thanks to his/her ability to reuse the skills previously learnt while learning how to ride a bicycle.

The field of Lifelong Machine Learning (LML) studies systems that can learn many tasks over a lifetime from one or more domains. These systems should efficiently and effectively retain the knowledge they have learned and use that knowledge to more efficiently and effectively learn new tasks (D. L. Silver, Yang, & Li, 2013). Lifelong machine learning is still a large, open problem and is of great importance to the development of AGI. Many of the tasks that an agent can encounter in its lifetime can be decomposed into sub-behaviours or skills. For example, consider making an omelette. An agent needs to crack the eggs, whisk them and put them in a pan. However, if a new task appears, such as making a cake for example, the agent can partially reuse the knowledge it gained from cooking the omelette to learn this new task faster. Reinforcement Learning (RL) provides a way to learn skills from experience. It is a branch of ML well suited for various tasks in lifelong learning. RL techniques have been used in a wide variety of areas with outstanding results. Near human-level performance has been achieved in programs that can play games such as checkers (Samuel, 1959), backgammon (Tesauro, 1994) or Go (D. Silver et al., 2016; D. Silver et al., 2017). RL has also been used in the field of robotics (Kerzel, Mohammadi, Zamani, & Wermter, 2018; Kober, Bagnell, & Peters, 2013; Luo, Kasaei, & Schomaker, 2020).

Although at first, RL may seem like a plausible approach to solve the AGI problem (Hutter, 2007), it is prone to suffer from the *curse of dimensionality*. The curse of dimensionality is a term coined by Bellman (1961) to refer to the exponential increase in the state-space with each additional variable or dimension that describes the problem. It seems unlikely that complex problems can be described by a small number of variables. Fortunately, the real world is highly structured and most parts are independent of most other parts. Many large complex systems in nature exhibit hierarchical structure that allows them to be broken down into smaller sub-problems. The sub-problems, being smaller, are often solved more easily. A solution to the original problem can be obtained by the combination of the solutions to the smaller sub-problems. This approach of decomposing a complex task can have significant effects in the time and space complexity for both learning and execution. By introducing a **hierarchical structure** to RL (Barto & Mahadevan, 2003), we can reduce complex problems to a more manageable size. As stated by Hengst (2012), "*Hierarchical Reinforcement Learning (HRL) rests on finding good re-usable temporally extended actions that may also provide opportunities for state abstraction*".

In this work we focus on finding ways to maximise the capacity of an agent to effectively and efficiently learn new tasks throughout its lifetime. By leveraging the potential of HRL, we make an agent able to reuse previously learnt skills when trying to solve a new task. This re-usability of skills is what makes lifelong learning viable. Learning to solve a new task from scratch requires the allocation of a certain amount of computing resources. However, if the new task's action space overlaps with previously learnt tasks' action spaces, previously acquired knowledge can be reused and thus, the allocation for new computing resources is lower. This can make the total amount of resources that an agent would need throughout its lifetime orders of magnitude lower. To test and develop this work, we will use a custom-made environment. The environment consist of a hypothetical domestic robot that needs to solve common household tasks. For example, some of those tasks can be making tea for the home owner or making soup.

1.1 Research Questions

With the work presented in this thesis, we aim to provide an answer to the following research questions.

- Can we design a system that can learn new tasks throughout its lifetime and effectively and efficiently retains and reuses previously gained knowledge to solve new problems?
- How does this system perform in a specific scenario where a simulated household robot needs to solve common tasks?

- How does this system perform compared to a standard reinforcement learning algorithm such as Q-learning?
- Is such a system a suitable option for a lifelong learning scenario, such as the household robot environment?

1.2 Thesis Outline

This thesis is structured in the following manner. In Chapter 2 a theoretical background suitable for this thesis will be presented. Basic principles of reinforcement learning are presented, as well as principles of hierarchical reinforcement learning and past and current approaches. Also, some information about lifelong machine learning literature is summarized. In Chapter 3 there is a description of the functionality and dynamics of the simulated environment created. Additionally, a detailed description of the architecture of the proposed solution is given. A description of the most relevant experiments and their results is presented in Chapter 4. In Chapter 5, conclusions drawn from the experiments are presented, as well as potential shortcomings of this approach. The chapter ends with some ideas for future work and a brief conclusion.

Chapter 2

Theoretical Background

There are some concepts that the reader should be familiarised with in order to fully understand the work described in the following chapters. These concepts are briefly explained in this chapter in order to contextualize and help the reader understand the rest of this writing.

2.1 Reinforcement Learning

"Reinforcement learning is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods" (Sutton & Barto, 2018). In contrast to other machine learning paradigms, such as supervised learning, the learner is not told the correct answer (which actions to take) but it has to learn by interacting with the environment. Learning, then, becomes a process of discovering which actions yield the most reward. A reinforcement learning system contains two elements: an agent (or agents) and an environment. An agent must be able to perceive the state of the environment to some extent and must be able to take actions that affect the environment. The agent must also have a goal or goals relating to the environment. Beyond these two key elements, one can identify three main sub-elements: a *policy*, a *reward function* and a *value function*.

The *policy* defines the agent's way of behaving at any given time. It is a mapping from states of the environment to actions to be taken when in those states.

The *reward function* defines the goal of a reinforcement learning problem. After every single action, the environment sends a numeric signal to the agent. The objective of the agent is to maximize the total reward it receives over the long run.

The *value function* indicates what is good in the long run. For instance, the value of a state is the total amount of reward that the agent can expect to obtain starting from said state.

2.1.1 Finite Markov Decision Processes

A Markov Decision Process is intended to include the 3 main sub-elements that constitute a reinforcement learning system: states, actions and goals. Markov decision processes or MDPs are a formalization of sequential decision making, where actions not only affect immediate rewards, but also future states and by extension, future rewards. The agent and the environment continually interact with each other for a

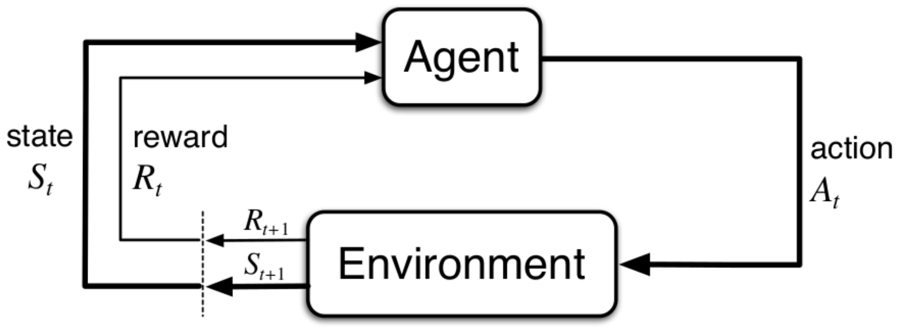


Figure 2.1: Diagram of the interaction of an agent and its environment in an MDP. Picture taken from (Sutton & Barto, 2018)

sequence of discrete time steps $t = 0, 1, 2, \ldots$. At any given time step t, the agent observes the environment's state $S_t \in \mathcal{S}$, and then chooses to take an action $A_t \in \mathcal{A}$. In the next time step, $t + 1$, the environment sends to the agent a numerical reward signal $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. The agent then perceives the environment's state $S_{t+1} \in \mathcal{S}$. A diagram of the interaction can be seen in Figure 2.1. The agent and the environment thereby produce a sequence in the form of:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.1}$$

In a *finite* MDP, the sets of states, actions and rewards ($\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$) all have a finite number of elements. The random variables R_t and S_t have well defined discrete probability distributions dependent only on the previous state and action. For particular values of these random variables $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r \mid s, a) \doteq Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right\}, \tag{2.2}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$. The function p defines the dynamics of the MDP. The dot over the equals sign in the equation reminds us that it is a definition (in this case of the function p) rather than a fact that follows from previous definitions. The '|' in the middle of it comes from the notation for conditional probability, but here it just reminds us that p specifies a probability distribution for each choice of s and a, that is,

$$\sum_{s' \in \mathcal{S}} \sum_{r' \in \mathcal{R}} p(s', r \mid s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.3}$$

Goals and rewards

As mentioned earlier, in a reinforcement learning setup, an agent will try to maximize the reward over the long run. To formalize it mathematically, it needs to maximize the *expected return* G_t, where the return can be written as:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.4}$$

In some cases, the interaction between the agent and the environment goes on without limit ($T = \infty$). For said cases, Equation 2.4 could easily go to infinite.

For that particular reason, an additional term is added to that equation to make it mathematically simpler. With the addition of a *discount rate* the expected return G_t becomes bounded as $t \to \infty$:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.5}$$

where $0 \leq \gamma \leq 1$ is the discount rate.

Policies and value functions

In nearly all reinforcement learning algorithms, the agent needs to estimate how good it is for itself to be in a given state (or how good it is to take certain action in a given state). We can define the "how good" in terms of the expected return. The *value function* gives the notion of how good a certain state (or state-action pair) is. However, since the future rewards that the agent will receive depend on what actions it takes, value functions are defined with respect to behaviours, called *policies.* A policy is formally defined as,

$$\pi(a \mid s) = Pr\{A_t = a \mid S_t = s\}, \text{ for all}\, s \in \mathcal{S} \,\text{and}\, a \in \mathcal{A}, \tag{2.6}$$

or in other words, if the agent is following policy π at time t, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$. Essentially, π specifies a probability distribution for each choice of s, and therefore satisfies

$$\sum_{a \in \mathcal{A}} \pi(a \mid s) = 1, \text{ for all}\, s \in \mathcal{S} \tag{2.7}$$

The *state-value* function for *policy* π, denoted as v_π, is the expected return starting from any state s and then following policy π. Formally, v_π is defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right], \text{ for all } s \in \mathcal{S} \tag{2.8}$$

Similarly, the *action-value* function for policy π, denoted as q_π, is the expected return starting from any state s, taking action a and then following policy π. Formally, q_π is defined as:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a \right]. \tag{2.9}$$

Value functions satisfy recursive relationships. That is why we can also write the *state-value* function $v_\pi(s)$ as:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \Big] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \text{ for all } s \in \mathcal{S}, \end{aligned} \tag{2.10}$$

and similarly for the *action-value* function $q_\pi(s,a)$ as:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\ &= \sum_{s'}\sum_{r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a' \mid s')\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s', A_{t+1} = a']\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a')\right] \end{aligned} \tag{2.11}$$

Equations 2.10 and 2.11 are known as the *Bellman equation* for v_π and q_π respectively. It expresses a relationship between the value of a state and the values of its successor states.

Optimal policy

For finite MDPs, there is always at least one *optimal policy.* The optimal policy is the one that is always better than or equal to all other policies. The optimal state-value function, denoted as v_*, is defined as

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}, \tag{2.12}$$

and similarly, the optimal action-value function:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s), \tag{2.13}$$

The optimal value function v_* must satisfy the self-consistency condition given by the Bellman equation (2.10). We can write the Bellman equation for v_*, or the *Bellman optimality equation*

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right] \tag{2.14}$$

and the Bellman optimality equation for q_*:

$$q_*(s,a) = \sum_{s',r|s,a} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{2.15}$$

We refer to a beautiful and simple explanation of the Bellman optimality equation, present in the work of Sutton and Barto (2018):

"*Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state*"

2.1.2 Exploration–Exploitation Dilemma

Unlike other kinds of learning, one challenge in reinforcement learning is the trade-off between exploration and exploitation. To maximize the reward, the agent must take actions that, in the past, were found to be effective. That is called the exploitation phase. However, it also needs to try actions that it has not selected before in order to discover the most effective actions. An agent must *exploit* what it has learned to obtain reward, but it must also *explore* so it learns what the best possible action in each state is.

One commonly used approach to deal with this exploration–exploitation dilemma is the *epsilon greedy* or ϵ*-greedy* method. With this approach, the probability of taking an action following the optimal policy π_* is $1 - \epsilon$. A way that is usually implemented in reinforcement learning algorithms is shown in Algorithm 1.

Algorithm 1: ϵ-greedy algorithm

```
In each time step do:
Generate a random number n ∈ $[0, 1]$
if n $< \epsilon$ then
    action $\leftarrow a \sim \mathcal{U}\{\mathcal{A}\}$        /* sample a random action */
else
    action $\leftarrow argmax_{a \in \mathcal{A}} Q(s, a)$
end
Decrease $\epsilon$ following update rule (Optional)        /* e.g. Eq. 2.16 */
```

To achieve a faster convergence, the value of ϵ can be slowly decreased during training. That allows the agent to perform an intense exploration phase at the beginning, while it does a more exploit-oriented phase towards the end. An update rule could be simply $\epsilon \leftarrow \eta\epsilon$, where $0 \leq \eta \leq 1$ is a parameter called the *decaying rate*. However, this update rule can be defined as anything and another formula that is commonly used is:

$$\epsilon = end + (start - end)e^{-steps/\eta}, \tag{2.16}$$

where the *start* and *end* are parameters that define the upper and lower limits of ϵ respectively, and *steps* is the total number of steps since the beginning of the training. This method generates an exponential decay of ϵ, while also bounding it to a minimum and maximum values. For instance, if the *end* value is set to 0.1, the agent will always take an exploratory action with at least a 10% probability. Similarly, if the *start* value is set to 0.9, the agent will explore with at most 90% probability.

2.1.3 Q-Learning

One of the early breakthroughs in reinforcement learning, and still used nowadays, is the algorithm known as *Q-learning* (Watkins, 1989). It is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right], \tag{2.17}$$

where $0 \leq \alpha \leq 1$ is the *learning rate*, a parameter that controls the speed at which Q converges.

Q-learning is an off-policy iterative algorithm that tries to find an action-value function Q, which directly approximates the optimal action-value function q_* (Equation 2.15). It has been shown to converge to q_* with probability 1. The simplest method to store the calculated Q-values is by using a look-up table. An entry $Q(s, a)$ exists for each pair of state-action values. These values are dynamically updated following the update rule shown in equation 2.17. Although it is a good approach for simple problems, it starts to fail when the state/action spaces get big. Having the need for a single entry for each state-action pair becomes too expensive computationally.

Function Approximators are a set of techniques that aim to solve this scalability issue. Instead of storing a precise value for each state-action pair like in look-up tables, function approximators approximate the $Q(s, a)$ value. There are different types of function approximators such as Linear combination of features, Decision trees, Nearest Neighbour, etc. Nonetheless, by far the most widespread function approximators nowadays are Artificial Neural Networks or ANNs.

2.1.4 ANNs as Function Approximators

The theory behind Neural networks is extensive and out of the scope of this work. We refer to works such as McClelland, Rumelhart, et al. (1986) or Goodfellow, Bengio, and Courville (2016), which cover this field thoroughly. However, we will present a brief review of one of the simplest neural networks, the Multi-Layer Perceptron or MLP.

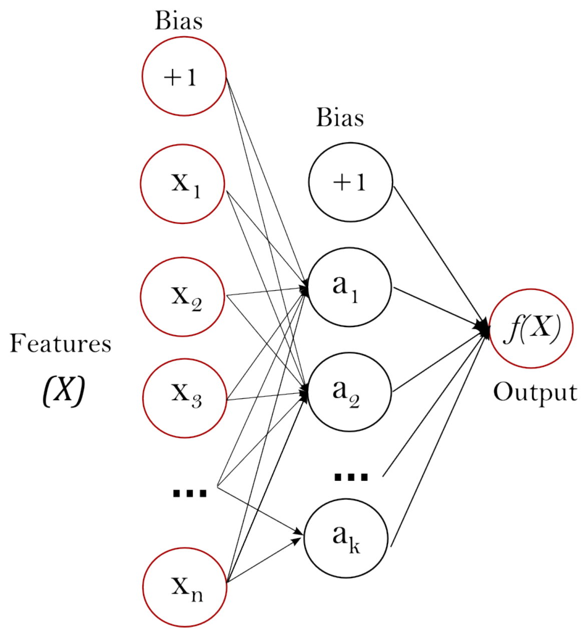


Figure 2.2: Multi-layer Perceptron. Figure taken from (Scikit-learn, 2019)

The Multi-Layer Perceptron (Figure 2.2) is an extended version of the Perceptron presented by Rosenblatt (1962), and first studied by Rumelhart, Hinton, and Williams (1985). In a Perceptron, a certain number of inputs (or input layer) are connected to a certain number of outputs (output layer), and the strength or *weight* of these connections is what determines the output values. The peculiarity of a Perceptron is that the output can be modified by just changing the weights.

Each unit in an Perceptron is called a neuron. A neuron is basically a computational unit. Its output value y_j is the result of applying a non-linearity to a weighted sum of the inputs. It can be formally defined as:

$$y_j = \phi\left(\boldsymbol{w}_{ji} \cdot \boldsymbol{x}_i + b_j\right) \tag{2.18}$$

where $\boldsymbol{x}_i$ is a vector with the values of the neurons in the i^{th} layer, $\boldsymbol{w}_{ji}$ is a vector with the weights from the i^{th} layer to the neuron j, b_j is a bias and ϕ is a non-linear operation such as a sigmoid function. Multi-layer Perceptrons are simply Perceptrons with at least one additional layer in between the input and the output. Those additional layers are called *hidden layers*. Multi-layer Perceptrons with at least 1 hidden layer with enough units have been shown to be able to approximate any type of function.

Since the output of an MLP can be modified by just changing the weights, we need a method for finding the optimal weights that can make an MLP approximate a given function. Nowadays, most artificial neural networks use the method of *back-propagation* to find the optimal weights. Back-propagation (Rumelhart, Hinton, & Williams, 1986) is a gradient descent technique that tries to minimize a cost function or a *loss*. One can choose any kind of cost function, depending on the type of model and data that is being used. Some cost functions can work significantly better with certain types of data distributions and models. As an example, one of the most commonly used cost functions is the Mean Squared Error (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2, \tag{2.19}$$

where n is the number of predictions generated from a sample of n data points on all variables, Y is the vector of observed values of the variable being predicted and $\hat{Y}$ are the predicted values.

Once the loss has been computed for a particular batch of data, the weights can be updated following:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \Delta\boldsymbol{w}, \tag{2.20}$$

where $\boldsymbol{w}$ are the weights of the MLP and $\Delta\boldsymbol{w}$ represents the update which, in its simplest form (Stochastic Gradient Descent), can be written as:

$$\Delta\boldsymbol{w} = \alpha\nabla E(\boldsymbol{w}), \tag{2.21}$$

where α is the learning rate and $\nabla E(\boldsymbol{w})$ is the gradient of the cost function with respect to the weights.

2.1.5 Experience Replay

The use of ANNs as function approximators has generated some of the biggest successes in the field of reinforcement learning. Works such as the studies conducted by Mnih et al. (2013), Mnih et al. (2015) were a breakthrough in the field. In said work, they were able to create an agent capable of playing over a 100 different Atari games, without the need to use handcrafted features for any of them. One particular component that was key to their success is the *experience replay*.

Experience Replay (ER) is a method first studied by Lin (1992). This method saves the agent's experience at each time step and stores it in a replay memory that can be accessed any time to perform the weight updates. The common implementation of experience replay goes like this. After an agent performs a certain action A_t in a state S_t, and the environment returns the reward signal R_{t+1} and the next state S_{t+1}, the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ is added to the replay memory. This replay memory accumulates many experiences of the same environment. At each time step, several Q-learning updates —in the form of a mini-batch update— are performed using data randomly sampled from the replay memory.

The effects of sampling experiences uniformly at random are very consequential. In a normal RL setup, consecutive experiences are strongly temporally correlated. However, gradient descent techniques —the de facto technique to train ANNs— do not perform as well when the data used is correlated. Randomly sampling the data reduces the variance of the updates because consecutive updates are not correlated with one another. Moreover, ER got rid of one source of instability by eliminating the dependence of successive experiences on the current weights.

2.2 Hierarchical Reinforcement Learning

Many complex systems in nature exhibit hierarchical structure. In this context, hierarchical structure means that systems can be usually decomposed into smaller sub-systems, which can also be decomposed into smaller sub-systems and so on. A common approach for problem-solving is *decompose and recombine* or as the common saying goes *divide and conquer*. Big complex problems can be tackled by solving smaller sub-problems, which tend to be easier to solve. A solution to the original large problem can be found by recombining the solutions to the smaller sub-problems. An additional reason that hierarchy can simplify problem-solving is that the same sub-problem or task can appear in different contexts. If a learning agent is not aware of this, it will relearn the same task in various contexts. Optimally, the agent only needs to learn the task once and then it can reuse that knowledge in other contexts. Bringing a hierarchical structure to reinforcement learning is the field of study of *Hierarchical Reinforcement Learning*.

"In many ways hierarchical reinforcement learning (HRL) is about structuring reinforcement learning problems much like a computer program where subroutines are akin to subtasks of a higher-level reinforcement learning problem. Just as the main program calls subroutines, a higher level reinforcement problem invokes subtasks." (Hengst, 2012)

There are numerous works on the field of HRL. Since earlier works such as, Dayan and Hinton (1993), Sutton, Precup, and Singh (1999), Wiering and Schmidhuber (1997) or Dietterich (2000), to more recent publications in this field such as Levy, Platt, and Saenko (2018), Nachum, Gu, Lee, and Levine (2018), Niel and Wiering (2018), Vezhnevets et al. (2017) or Watters, Matthey, Bosnjak, Burgess, and Lerchner (2019). One particular issue with most of these approaches is that the hierarchy needs to be defined a priori, and the success of the algorithm depends greatly on the correct division of the problem into appropriate sub-tasks. Some research (Hengst, 2002; Moerman, Bakker, & Wiering, 2007) tries to find ways of splitting a task into sub-tasks automatically. While these approaches lift the burden of having to predefine the hierarchy, they might be sub-optimal for a number of situations.

Most of the above mentioned algorithms try to solve a particular task and need to be retrained to solve a new task. Some reuse some elements, like the vision module in Watters et al. (2019). Others can train shared sub-policies that can solve multiple tasks (Frans, Ho, Chen, Abbeel, & Schulman, 2017). A very interesting work was done by Tessler, Givony, Zahavy, Mankowitz, and Mannor (2016), where they present an HRL approach to lifelong learning, in which sub-policies or *skills* are partially reused to solve new tasks. They propose two methods of achieving effective and efficient skill re-utilisation and storing.

2.3 Lifelong Machine Learning

Lifelong machine learning is a sub-field of machine learning that focuses on the study of systems that can learn continually during the span of their lifetimes. As stated by D. L. Silver et al. (2013):

"*Lifelong Machine Learning, or LML, considers systems that can learn many tasks over a lifetime from one or more domains. They efficiently and effectively retain the knowledge they have learned and use that knowledge to more efficiently and effectively learn new tasks*".

It is believed that the following are essential elements for an LML agent: (1) the *retention* of learned task knowledge; (2) the selective *transfer* or use of prior knowledge when solving new tasks; and (3) a *systems* approach that ensures the effective and efficient interaction of the aforementioned *retention* and *transfer* elements.

In this context, talking about knowledge *retention* is done with a knowledge representation perspective. Any learned knowledge can be represented in various forms. The simplest form can be simply storing training examples. Storing the raw training data has the advantage of accuracy and purity of the knowledge (knowledge retention). However, it is likewise inefficient due to the large amount of storage that it requires. Alternatively, a representation of an accurate hypothesis developed from the training examples can be stored. The advantages of representational knowledge are its small size compared to the space required for the original training data and its ability to generalize beyond the training examples.

Talking about knowledge *transfer* is done from the machine learning perspective. Representational transfer involves the assignment of a known task representation to a learning system with a new target task. By doing so, the new model is initialized in a particular region of the hypothesis space of the modeling system. Representational transfer often reduces the training time of the new model (efficient) with no significant loss in the generalization performance of the resulting hypotheses. In contrast to representational transfer, functional transfer employs the use of implicit pressures from training examples of related tasks, the parallel learning of related tasks constrained to used a common internal representation, or the use of historical training information from related tasks. These pressures reduce the hypothesis space in which the learning system performs its search. This other form of transfer tends to develop a more accurate hypothesis (effective).

The *systems* approach emphasizes the necessary interaction between knowledge *retention* and *transfer* learning, and that lifelong machine learning is not just an algorithm. LML can benefit from new research on learning algorithms and training techniques, but it also involves the retention and organization of knowledge.

Advances in the field of LML can be applied to systems such as autonomous robots and intelligent web agents. Robots such as the ones sent on exploratory missions in space or under the sea must have the ability to learn new things and recognize new objects after they have been deployed, and must take decisions during extended periods of time within a constantly changing environment. Same goes for web agents and personal assistants. These agents need to be able to adapt to changes in people's behaviours. The ability to retain an reuse knowledge is very attractive to researchers designing such systems.

Chapter 3

Methodology

In the previous chapter, concepts such as Q-learning, experience replay and Hierarchical Reinforcement Learning were briefly introduced to the reader.

In this chapter, we present a new approach that could potentially thrive in a lifelong machine learning scenario. Such a system was designed by combining several reinforcement learning techniques. The architecture and the training process of this novel approach will be explained.

Moreover, since this approach is oriented towards lifelong learning, a new simulated environment was created. The simulated environment devised is one that is able to represent a lifelong learning scenario where an agent is able to solve several tasks, and it is one where new unseen tasks can always occur.

3.1 Environment

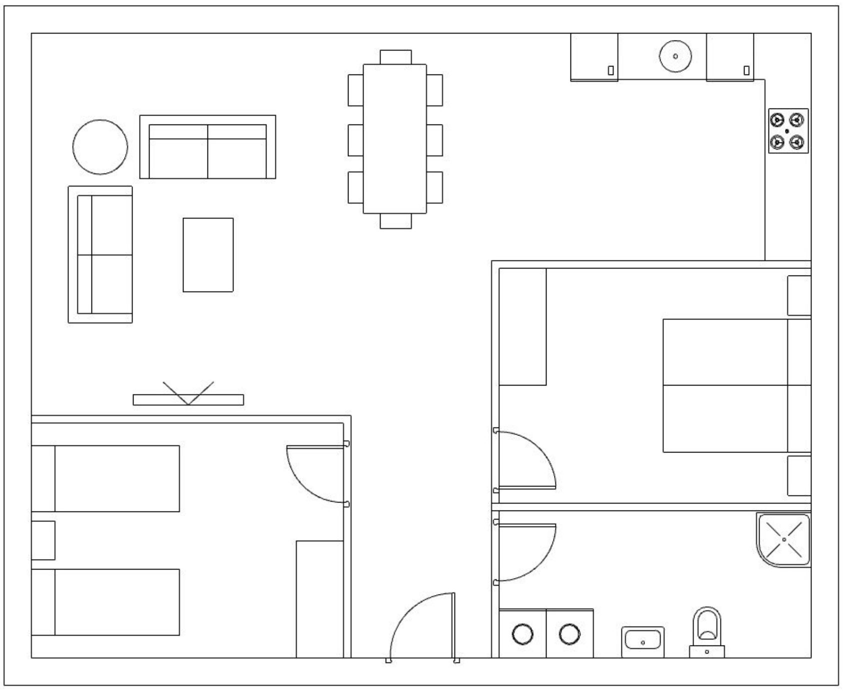
As it was briefly discussed in Chapter 2, the environment is a key component of any reinforcement learning system. It contains everything that the agent can sense and interact with. Having a well-defined environment is a key component for finding a successful solution. If an environment provides very little information about its state, it might be impossible for the agent to solve any task. Likewise, if the reward function is erratic or inconsistent, it can confuse any learning algorithm and the agent might find it impossible to solve the task too.

A completely new simulated environment was created for this thesis. It tries to capture the idea of a hypothetical household assistant robot that needs to do common household chores. An RL agent can interact with this environment as seen in Figure 2.1 and thanks to the reward function it can learn to solve predefined household tasks.

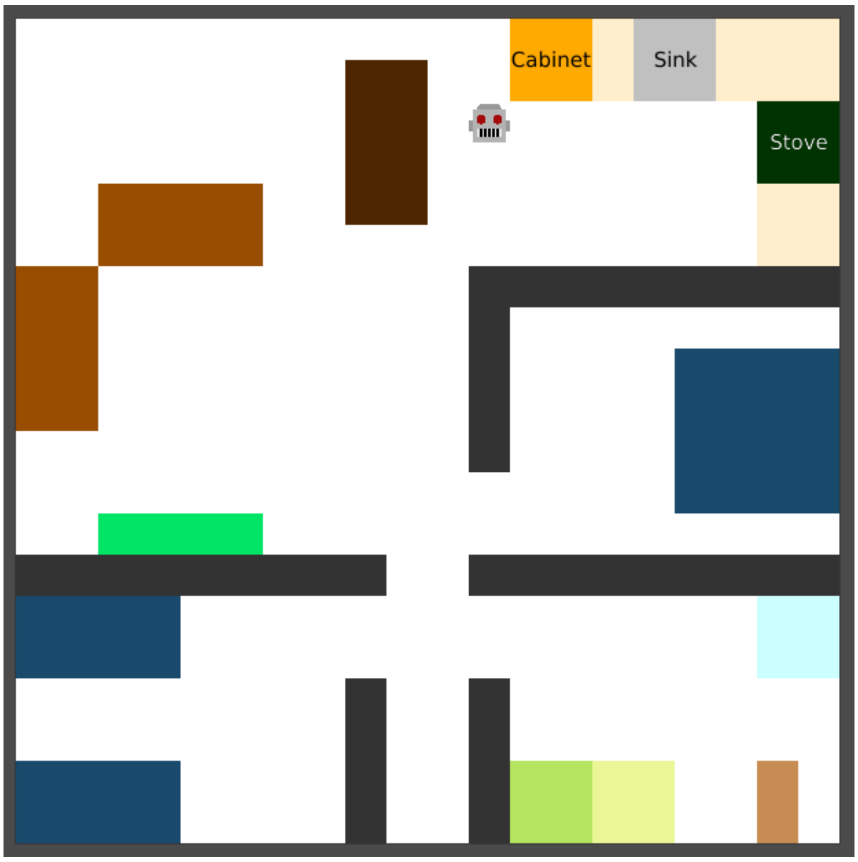
3.1.1 Layout and Dynamics

First, a hypothetical household blueprint was designed (Figure 3.1a). It contains all the common components of a regular household such as tables, beds, kitchen appliances, etc.

Based on this blueprint a simulated RL environment was created. It uses the OpenAI's Gym framework (Brockman et al., 2016). The Gym framework is a building block frequently used within the reinforcement learning community. It was one



(a)



(b)

Figure 3.1: (a) Blueprint of a hypothetical regular household. (b) Simulated environment used in this work

of the first frameworks that allowed users to: a) Access a repository of different environments to easily develop, test and compare new algorithms, b) Tools to create new RL environments that have a common, easy-to-use interface. This framework makes it extremely easy for one researcher to use an environment created by another researcher. That is the main reason for choosing the Gym framework for this thesis.

The simulated environment created is a grid-like map that represents Figure 3.1a. A screen capture of it is shown in Figure 3.1b. The dimensions are 20×20 unit-squares. Each piece of furniture or appliances has a distinct color, while the walls are dark grey. Each time step, the robot can freely move and perform actions inside the white area. The environment is set to terminate each episode after 1000 time steps.

As it was introduced in Chapter 2, an agent must have a way to perceive the state of the environment. In order to have a simple state space, most of the state of the environment is represented as a binary-like type of question. For instance, one input to the agent indicates the state of the cabinet door with a binary input 0/1 for closed/open. All the possible information that will be available to the agent throughout the training is shown in Table 3.1. The last component in the table

Input	Min	Max
Position x	0	19
Position y	0	19
Cabinet door open	0	1
Has tea in inventory	0	1
Has soup jar in inventory	0	1
Has cleaning cloth in inventory	0	1
Has cleaning product in inventory	0	1
Has pasta in inventory	0	1
Has sauce in inventory	0	1
Has eggs in inventory	0	1
Has milk in inventory	0	1
Has pancake mix in inventory	0	1
Pasta drained	0	1
Whisked	0	1
Rinsed	0	1
Item cooked	0	1
Fire turned on	0	1
Tap open	0	1
Saucepan in hand	0	1
Frying pan in hand	0	1
Saucepan full	0	1
Heated up	0	1
Has boiling water	0	1
Stove cleaned	0	1
Task encoding ($n = \lceil \log_2 m \rceil$)	0	1

Table 3.1: Input features that the agent can access. Total of 29 inputs. (The task that needs to be done is binary encoded, so the number of task inputs n is defined by the total number of possible tasks m.)

(*Task encoding*) is not one, but several inputs. That is because the task that the agent needs to learn is encoded before presenting it to the agent. More will be discussed about tasks in Section 3.1.3, but the reason behind encoding it is because a learning system such as an MLP can wrongly infer that there is an ordering among the tasks, where actually the index numbering is just completely arbitrary.

In addition to the ability of perceiving the state of the environment, the agent must have the capacity to interact with it. That means that actions executed by the agent have a direct impact on the state of the environment. All the actions that the agent can execute are listed in Table 3.2. It is important to note that the actions will not always change the state of the environment. For example, if the agent tries to open a door while it is already opened, nothing will change in the state of the environment. Likewise, if the agent tries to open a door when it's not in front of one, the door will remain closed. Moreover, the state of the environment does not change unless the agent interacts with it or the episode ends. For instance, if the kitchen tap is open, it will remain like that until the agent closes it or the episode concludes.

Actions			
Move up	Open door	Fill	Heat up
Move down	Close door	Add & mix	Drain
Move left	Open tap	Turn on fire	Scrub
Move right	Close tap	Turn off fire	Rinse and dry
Get saucepan	Get tea	Get soup jar	Get pasta
Get sauce	Get cleaning cloth	Get cleaning product	Get eggs
Get milk	Get pancake mix	Get frying pan	Whisk
Flip			

Table 3.2: Possible actions that the agent can perform in this environment; Total of 29 actions.

3.1.2 Rewards

Another key component of any reinforcement learning system is the reward function. It is what enables the agent to learn how to solve a task. The agent receives a small negative reward for moving and a slightly lower negative reward for taking wrong actions and whenever it collides with an object or a wall. A positive reward of 100 is given whenever it manages to solve a particular task.

Reward type	Numerical value
Move	-0.1
Bump into a wall/furniture	-3
Inadequate action	-3
Adequate action	0
Task (intended) completed	100

Table 3.3: Reward function numerical values

Actions are considered inadequate when they're taken from a state of the environment that does not allow it. For instance, trying to open a door when the agent is not in front of it or trying to open the tap when it's already opened. Similarly, actions are considered adequate when they are taken from a state of the environment that allows it. For example, turning the stove on when the agent is in front of it and it was previously off.

It must be noted that it only receives a reward for solving a task if that task was the one that it was meant to solve from the beginning. A more in-depth explanation is given in Section 3.1.3, but mainly the agent should only be rewarded if it solves the task that it was asked for. If, for example, a user wants a cup of tea, the robot should only be rewarded if it makes a cup of tea. Solving a different task in that situation, for example making pancakes, would not yield a positive reward. Table 3.3 displays the kind of rewards and their numerical values.

3.1.3 Tasks

As it has been briefly introduced in previous sections, the goal of this reinforcement learning agent is to learn how to solve multiple tasks. A regular household has numerous tasks that are needed to be performed daily. However, to narrow the focus of this work, we have chosen a few tasks that belong to the kitchen domain only. Moreover, since the goal of the study is to develop algorithms that can re-use skills, most tasks are similar to others. That increases overlap and re-usability of skills.

All the tasks have a clear *action sequence* definition. For example, to have boiling water the agent needs to grab a saucepan, fill it up with water and then heat it up, all in that particular order. The definition of the action sequence is what determines how much a certain block of actions can be reused. For example, boiling water is composed of a sequence of actions and that block can be reused for various tasks. The action sequence of 2 relatively similar actions is shown in Figure 3.2, *Make soup* and *Make tea*. These two tasks share many actions and the sequence in which they should be taken. In this diagram, each yellow box is an action that the agent can take. The arrows represent the order in which those actions should be taken to solve the task. For example, to get tea, the robot needs to be in front of the cabinet, then open the door, then grab the tea and then it should close the door. Some action sequences are shared among these two tasks. For instance, it can be seen that for heating up a liquid inside a saucepan, the action sequence is exactly the same for both tasks.

In this reinforcement learning setup, we need a way to indicate the agent which task it needs to solve. The reason for that is because a typical reinforcement learning algorithm always tries to maximize the expected return. Since all tasks return the same reward, if the agent can get a positive reward for solving any task at any given time, the maximum expected return will always be obtained by solving the task that takes the least amount of time steps. The task that the agent needs to solve in a given episode, is presented to the agent as another variable of the state representation. The variable that gives that information was introduced in Table 3.1 with the name of *Task encoding*. It has that name because instead of just indexing all the possible tasks using natural numbers, the tasks are binary encoded before feeding it to the agent. In this situation, binary encoding simply means that instead of using a variable with a value of 5 to solve task n°5, the number 5 in binary representation ($0b101$) is fed to the agent. The reasoning behind it is because when

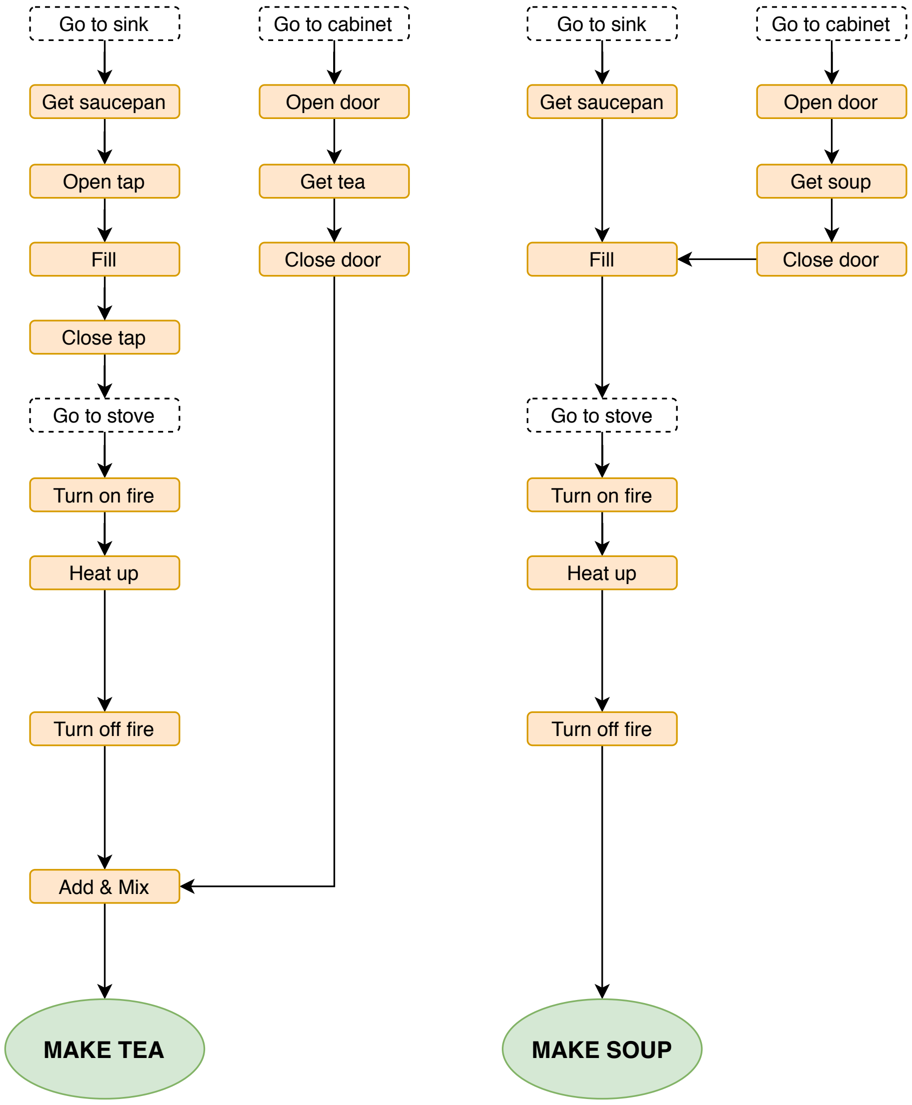


Figure 3.2: Diagram of the action sequence required to solve 2 similar tasks.

using a function approximator such as an MLP, the MLP might infer that there is an ordering among the tasks (task nº5 is more valuable than task nº3). Due to this issue, the number of state inputs for the agent is different based on the maximum number of tasks. Basically, the number of state inputs will be the minimum number of bits necessary to represent the total number of actions. It can be formally defined by the equation:

$$n = \lceil \log_2 m \rceil, \tag{3.1}$$

where n is the number of state inputs and m is the total amount of tasks (including the "do nothing" task represented by the 0).

There are six different tasks in total that can be solved by the agent. They are listed in Table 3.4 and the detailed action sequence of each one of them can be found in Appendix A. With the exception of the *Clean stove* task, the rest of the tasks share a similar structure. For instance, in order to cook something you always need to go to the cabinet, open the door, get the ingredients and then close the door.

The tasks proposed have different levels of difficulty. The difficulty of a given

Task
Clean stove
Make tea
Make soup
Make pasta
Make omelette
Make pancakes

Table 3.4: Possible tasks in the environment

task refers to the amount of actions that are required to solve it, as well as the time dependency that exists between those actions. *Make pasta* for example would be a high difficulty task, since it is composed of many actions that are time dependant (The agent can only drain the pasta after it's fully cooked). Meanwhile, *Clean stove* is a low difficulty task since it only takes 6 primitive actions to be solved.

3.2 Model

In this section, we describe the architecture of the proposed model, the different building blocks and the training method. This approach is inspired by the work of Frans et al. (2017), although there have been some modifications and new additions that are potentially better suited for lifelong learning.

3.2.1 Architecture

The model consists of a *master policy* θ and one or more *sub-policies* $\phi_1, \phi_2, \dots, \phi_K$. Each policy has its own independent MLP. Each sub-policy can choose to take any primitive action $A_t \in \mathcal{A}$ and all of them have access to all the components that make a full observation $S_t \in \mathcal{S}$. The master policy also shares the same observations. However, the master policy's goal is to choose the index $k \in \{1, 2, \dots, K\}$. Choosing the index means which sub-policy to follow in a given moment. For example, given

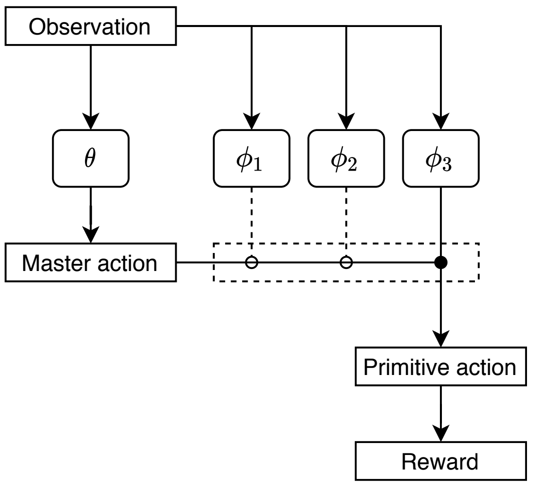


Figure 3.3: Model's architecture diagram

that the agent is in a certain state s_t, the master policy π_θ would yield an action k_t. Then for s_t, the agent would take the primitive action a_t yielded by the sub-policy π_{ϕ_k}. Figure 3.3 describes the model's behaviour when interacting with the environment. The idea behind this approach is that a given task can be solved by simply applying a combination of the sub-policies. This approach tries to maximize the re-usability of skills by letting the agent solve a task by using different policies at different stages of the execution of a task. For instance, a policy could represent the skill of opening a door, picking up an item and then closing the door. If this skill is needed to solve other tasks, then there is no need to "learn it" anymore.

3.2.2 Task-Specific Experience Replay

Experience replay is an element that enhances the learning process of a reinforcement learning algorithm. By eliminating the temporal correlation among experiences, the algorithm has a faster and more stable learning process (see Section 2.1.5).

In this work, we use experience replay with a particular approach. For every task that the agent needs to learn we create its own experience replay buffer. If for example there are six different tasks available in the environment, there will be six different experience replay buffers, each containing only transitions corresponding to a particular task. Figure 3.4 shows a diagram of how the task specific experience replay buffers work. Experiences are only saved to the corresponding ER. When training the agent to solve a specific task, the experiences are sampled from the corresponding experience replay buffer only. Having a single experience replay buffer for each task allows the system to train on previously explored tasks without the need of taking exploration steps again.

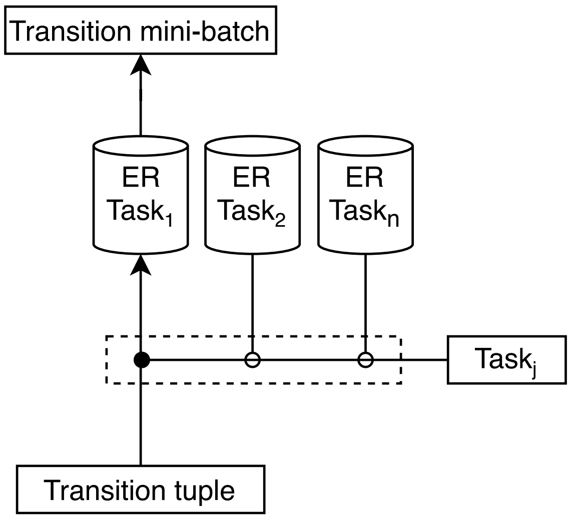


Figure 3.4: Task-specific experience replays

3.2.3 Training Process

There are various ways to train this model. As such, different variations in the learning process were conducted. However, we will only present the one with the best and most consistent results. It is important to note that, although this architecture can use any kind of RL algorithm, Q-learning is used to train all the policies.

Algorithm 2: Training algorithm

```
for $c = 0, 1, \ldots, C$ do                                /* C=Number of cycles */
    Sample task $T \sim P_T$        /* From possible tasks in experiment */
    Reset $\theta$                   /* Reset the weights of master policy */
    for $i = 0, 1, \ldots, I_\theta$ do            /* $I_\theta$=number of master updates */
        Update $\theta$ using a sampled mini-batch from $\mathrm{ER}_\theta^{task}$
    end
    for $i = 0, 1, \ldots, I_\phi$ do        /* $I_\phi$=number of sub-policy updates */
        Sample 1 transition from the $\mathrm{ER}_\theta^{task}$, where k=master action
        Update $\phi_k$ using a sampled mini-batch from $\mathrm{ER}_\phi^{task}$
    end
    Collect experiences for $N$ timesteps using $\epsilon$-greedy actions for master
      and sub-policy actions
    Store all transitions to $\mathrm{ER}_\theta^{task}$
    Store to $\mathrm{ER}_\phi^{task}$ only transitions where master action was not a
      exploratory step
end
```

The pseudo-algorithm for the training process is shown in Algorithm 2. It can be roughly divided into three blocks: task selection, policy updates and experience collection. The same process is repeated for a certain number of times and each iteration is called a cycle.

At the beginning of every cycle, a task is sampled from all the possible tasks that are relevant for the experiment $T \sim P_T$. This changes the focus of the model into task T for the rest of the cycle. Once the task has been selected, the master policy weights are reset and randomly initialized. By resetting the weights, the master policy is able to learn the optimal policy of one task without being influenced by the behaviour learnt on previously seen tasks. This action has more value in later stages of the process, when the agent tries to learn a whole new task but has the knowledge on how to solve other tasks already (e.g. it learns a third task after it already knew how to solve two tasks).

Next comes the *policy updates* block. In this block the MLPs are updated using experiences sampled from the experience replay buffers. There are two training loops, one for the master policy and another for the sub-policies. The idea is to train the master policy first, to an extent where θ is close to the optimal policy. That would mean that the master policy would select the best sub-policy in each scenario. Due to the nature of the training process (first swap focus, then train and then gather experiences) it was necessary to create a task-specific experience replay for the master policy $\mathrm{ER}_\theta^{task}$. This is just an implementation necessity and it is important to note that if, for example, the order of the blocks is swapped (first gather experiences then train), this task-specific master ER would not be needed and only one ER for the master policy would suffice. Once the master policy θ is near the optimal policy, the sub-policies are trained. I_ϕ iterations are performed following the same process. First a transition from the master experience replay $\mathrm{ER}_\theta^{task}$ is sampled. Then, based on that transition, the sub-policy ϕ_k is updated. The index k is obtained from the transition that was sampled before (remember that the master policy actions are the indexes k for the various sub-policies). The sub-policy ϕ_k is updated with a mini-batch sampled from the task-specific experience replay ER_ϕ^{task}, where the task indicated is the one that was selected previously in the task selection block.

The third block, *experience collection*, is very straightforward. A certain amount N of timesteps is gathered using the newly updated policies and the ϵ-greedy exploration method (as described in Algorithm 1 with ϵ-decay as described in Equation 2.16). There is one important addition that drastically improves the performance of the algorithm. Transitions are stored into the ER_ϕ^{task} only when the master policy action k was not a exploratory step (that is, master action k follows θ_*). The reasoning behind this idea can be explained with the following example. Imagine that for solving a certain task, the master policy is only using sub-policies ϕ_1 and ϕ_2. Then, if the master policy takes a random action (due to ϵ-greedy exploration) and selects sub-policy ϕ_3, it doesn't matter anymore what the sub-policy ϕ_3 does. Even if sub-policy ϕ_3 takes an action following its policy, since sub-policy 3 was not trained significantly before, the action will be far from optimal. In the best of cases, adding a transition to the ER_ϕ^{task} where the master policy took a exploratory step would mean just adding another random primitive action. Adding all the transitions adds noise into the ER_ϕ^{task} and delays (and can even derail) the learning process.

Training the system on a new task

The process discussed above is meant to train the agent on a certain number of tasks simultaneously. By the end of it, the agent will have learnt how to solve all the tasks that it was trained on, namely P_T. Although this first step would be always necessary for a real-life setup (a household robot is initialized with the knowledge of solving certain tasks), it is equally important that the agent is able to learn new tasks throughout its lifetime. The household robot should be able to learn a new task without forgetting the tasks it already knew. With this matter in mind, Algorithm 2 was slightly modified in order to adapt it to a set up where the agent can learn a new task without forgetting the previously learnt tasks. The training process for new tasks is shown in Algorithm 3. The new training process resembles the original one, but has several additions.

Firstly, the sub-policies $\phi_1, \ldots, \phi_K$ that were previously trained in other tasks are loaded. As we mentioned before, the agent does not start to learn from scratch this time. At the beginning of the process, the agent already has the knowledge that allows it to solve certain tasks. Therefore, it can use that knowledge to learn new tasks faster now.

Secondly, the experience replay buffers ER_ϕ^{task} that were filled in the previous phase are loaded too. The agent collected hundreds of timesteps worth of experience in the previous learning phase and that experience can be used now for the remind block, as will be explained below.

Lastly, the addition of a remind block. There is a risk that the agent might forget the tasks that it already mastered when training on a new task. To avoid this shortcoming, the agent is *reminded* of the old tasks every R number of cycles. Every R cycles, the sub-policies are updated using mini-batches sampled from the experience replay buffers that were loaded at the beginning.

Algorithm 3: Training on a new task

```
Load sub-policies $\phi_1, \ldots, \phi_K$
Load ER$_\phi^{task}$
for $c = 0, 1, \ldots, C$ do                          /* C=Number of cycles */
    Focus on new task $T$
    Reset $\theta$                  /* Reset the weights of master policy */
    for $i = 0, 1, \ldots, I_\theta$ do          /* I$_\theta$=number of master updates */
        Update $\theta$ using a sampled mini-batch from ER$_\theta^{task}$
    end
    for $i = 0, 1, \ldots, I_\phi$ do       /* I$_\phi$=number of sub-policy updates */
        Sample 1 transition from the ER$_\theta^{task}$, where k=master action
        Update $\phi_k$ using a sampled mini-batch from ER$_\phi^{task}$
    end
    Collect experiences for $N$ timesteps using $\epsilon$-greedy actions for master
      and sub-policy actions
    Store all transitions to ER$_\theta^{task}$
    Store to ER$_\phi^{task}$ only transitions where master action was not a
      exploratory step
    if $c \mod R = 0$ then        /* Do remind block every R cycles */
        for every task previously trained on do
            for $i = 0, 1, \ldots, I_\phi$ do
                Update $\phi_1, \ldots, \phi_K$ using sampled mini-batches from ER$_\phi^{task}$
            end
        end
    end
end
```

3.3 Q-learning Adaptation for Multiple Tasks and Lifelong Machine Learning

The standard Q-learning algorithm is not designed for multi-task training. It is not designed either for LML. It was necessary to introduce a few changes to the standard Q-learning algorithm to make it work on par with the HRL approach.

To adapt it to a multi-task scenario, the standard Q-learning algorithm is trained in the following manner. If, for example, the algorithm needs to be trained in tasks A and B, then at the beginning of each episode the focus is changed to one of the tasks randomly —lets say task A—. It collects 1 episode of experience and is also trained on that task. In the next episode, a new task is selected randomly —lets say B— and the process repeats. Algorithm 4 describes such a process.

To adapt the standard Q-learning algorithm to an LML situation, the system simply loads a previous model. If, for example, the system was already trained in tasks A and B, and now it needs to learn a new task C, the weights of the previous model —trained on A and B— are loaded. Starting from this point, the model is then trained normally following Algorithm 4, where $P_T = \{A, B, C\}$. By loading a previously trained model, the system doesn't have to start learning from scratch. Likewise, by introducing the new task C into P_T, the system is able to learn the new task C while not forgetting the previously learnt tasks (A and B).

Algorithm 4: Adapted Q-learning algorithm

```
for $c = 0, 1, \ldots, C$ do                          /* C=Number of episodes */
    Sample task $T \sim P_T$        /* From possible tasks in experiment */
    for each timestep in episode $c$ do
        Collect 1 tuple of experience
        Store experience tuple in ER
        Optimize model with sampled mini-batch from ER
    end
end
```

Chapter 4

Experiments and Results

The experiments performed in order to compare the proposed HRL algorithm and standard Q-learning are presented in this chapter. It should be noted that only the final experiments are presented. The algorithm described in Section 3.2 was achieved after an iterative process of trial and error where countless experiments were conducted before finding the right architecture and hyperparameters. The experiments presented below try to demonstrate how well the HRL approach performs compared to Q-learning in a lifelong learning scenario.

In a lifelong learning scenario, an agent needs to be able to learn new tasks as needed, while still being able to solve all the other tasks that were learnt before. We decided to design a series of experiments that best capture the idea of a household robot that keeps learning new tasks throughout its lifetime. We then compare the performance of both HRL and Q-learning approaches. The comparison between both approaches will be based on three metrics.

The first metric (*Time*) measures the execution time of the learning process. By definition, measuring the execution time of the learning process means how long it takes for the agent to learn the optimal policy π_*. This, however, is difficult to measure since it is not easy for us to know when — or if — the agent learns the optimal policy. A commonly used approach is to define a threshold that indicates that the agent has learnt a policy close enough to the optimal policy. For example, the agent always achieves a final episode reward greater than a certain value ($\sum_{k=0}^{T} R_{t+k+1} > \eta$), or it successfully solves the proposed task ρ out of the last 100 attempts. For all of the following experiments, we use a threshold $\rho = 75$. Therefore, the metric *Time* measures the time from start until the moment the agent solves the proposed task 75 out of the last 100 episodes. This metric is measured during the training phase of the algorithm

The second metric (*Steps*) measures the amount of time steps that were taken before reaching the aforementioned threshold ρ. Although quite similar to the time metric, this one is hardware-independent. The same model can learn in a shorter amount of time if it runs on better hardware, but it will roughly need the same amount of experience.

The third metric (*Accuracy*) measures the effectiveness of the algorithm. It measures the number of times the algorithm solved the task in 100 trials. To obtain this metric, the model is put in a testing mode. This means that the exploration rate ϵ is set to 0. Having a $\epsilon = 0$ makes the agent always follow the policy and thus, obtaining the maximum reward possible in each episode.

The fourth metric (*Retention*) indicates how many of the tasks that the agent already knew are still solvable by the agent. Knowledge retention is vital for LML and is a requirement that must be satisfied by any system that aims to be a successful LML approach.

4.1 Experiments

In this section we present the six experiments that were conducted in order to compare the proposed HRL algorithm and the standard Q-learning approach. In the first three experiments (4.1.1, 4.1.2, 4.1.3), both models are trained respectively on 1, 2 and 3 tasks simultaneously. Training a model on several tasks simultaneously represents the initialization of an agent, where a user already has a certain amount of tasks that need to be solved. In the following three experiments (4.1.4, 4.1.5, 4.1.6), both models are trained on one additional previously unseen task. The new tasks are learnt with different amounts of previous knowledge. Training a model on a new task with previous knowledge about other tasks represents a lifelong learning agent that needs to learn new tasks after it's been deployed.

All of the experiments presented in this section are obtained using the hyper-parameters shown in Appendix B. Each experiment was run three times and the results presented show the average and standard deviation of the three trials ($\mu \pm \sigma$). All of the experiments were run on the Peregrine high performance computing cluster, using Nvidia V100 GPUs.

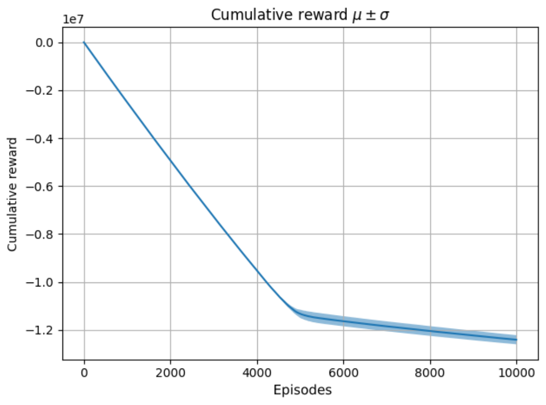
4.1.1 Learning One Task

The first experiment is also the one with the simplest conditions. The agent is trained in only one task with zero prior knowledge about the environment. In this case, a medium-high difficulty task was chosen (*Make pancakes*). Table 4.1 shows the metric values for both the HRL approach and the Q-learning approach, while the graphs created during training are shown in Figure 4.1. Although both approaches are able to learn the task successfully, it is clearly visible that the standard Q-learning approach performs better in this particular case. The metric *Time*, as well as the graphs show that the standard Q-learning approach takes less time to learn the task.

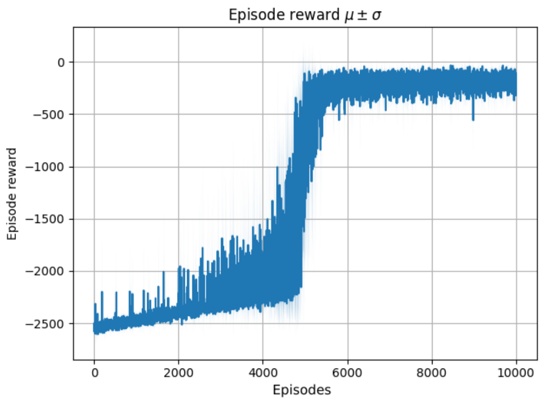
One of the main reasons for this gap is the experience distribution among sub-policies in the HRL approach. In this case, there are three different sub-policies. That fact entails that for the same amount of network updates and experience, the HRL approach needs to distribute the updates among the 3 sub-policies. This fact alone can result potentially in up to a $\times 3$ longer execution time. Moreover, the HRL approach also needs to train the master policy first at the beginning of each cycle, which consequently increases the total execution time.

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	3:32 ± 0:03	$4.88 \times 10^6 \pm 1.3 \times 10^5$	100%	-
Q-learning	0:34 ± 0:04	$4.98 \times 10^5 \pm 6.2 \times 10^4$	100%	-

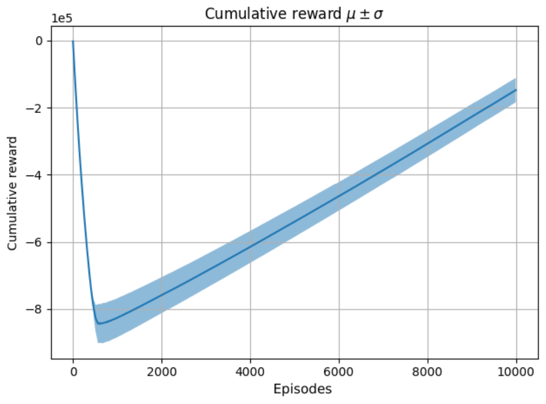
Table 4.1: Learning one task. Trained on *Make pancakes* task



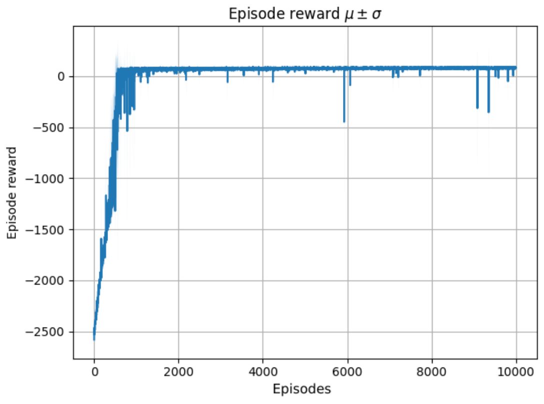
(a) Cumulative reward (HRL)



(b) Episode rewards (HRL)



(c) Cumulative reward (Q-learning)



(d) Episode rewards (Q-learning)

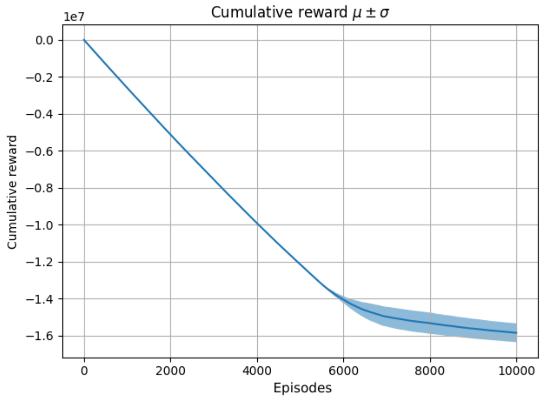
Figure 4.1: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Task: *Make pancakes*

4.1.2 Learning Two Tasks Simultaneously

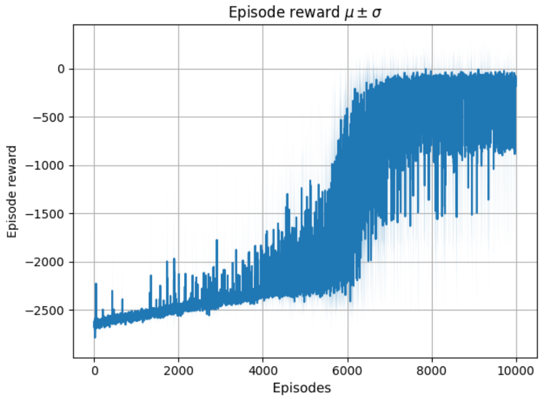
In this experiment both approaches are trained on two different tasks: *Make pasta* and *Make tea*. *Make pasta* is a high difficulty task while *Make tea* is a medium-low difficulty task. The HRL agent is trained using Algorithm 2. Table 4.2 shows the metric values for both the HRL approach and the Q-learning approach, while the graphs created during training are shown in Figure 4.2.

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	4:23 $\pm$ 0:20	$6.05 \times 10^6 \pm 4.7 \times 10^5$	100%	-
Q-learning	2:06 $\pm$ 1:39	$1.79 \times 10^6 \pm 1.5 \times 10^6$	100%	-

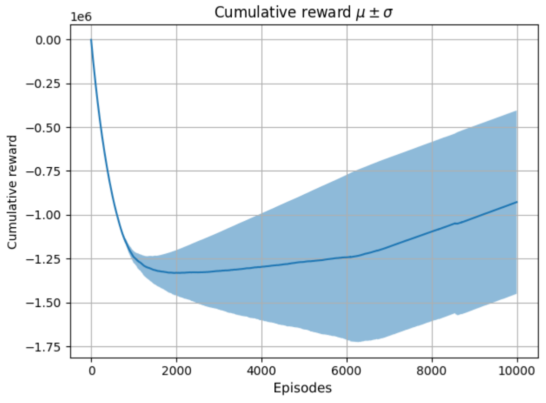
Table 4.2: Learning two tasks. Trained on *Make tea* and *Make pasta* tasks



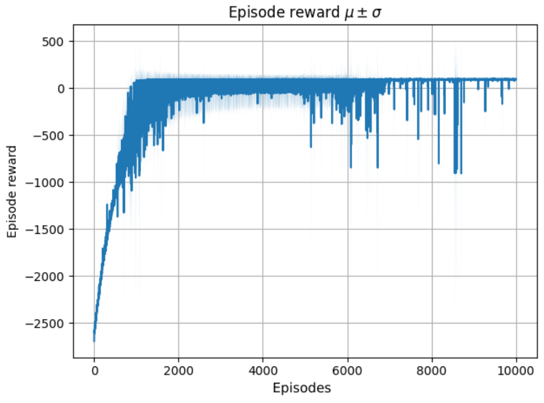
(a) Cumulative reward (HRL)



(b) Episode rewards (HRL)



(c) Cumulative reward (Q-learning)



(d) Episode rewards (Q-learning)

Figure 4.2: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Tasks: *Make tea* and *Make pasta*.

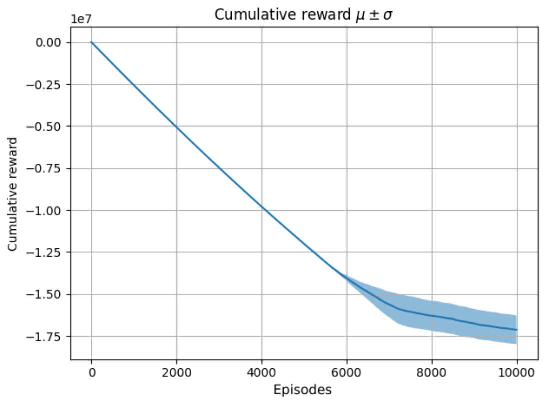
4.1.3 Learning Three Tasks Simultaneously

In this experiment, both approaches are trained on three different tasks simultaneously: *Make pasta*, *Make tea* and *Make omelette*. The HRL system uses Algorithm 2. The results are shown in Table 4.3 and Figure 4.3.

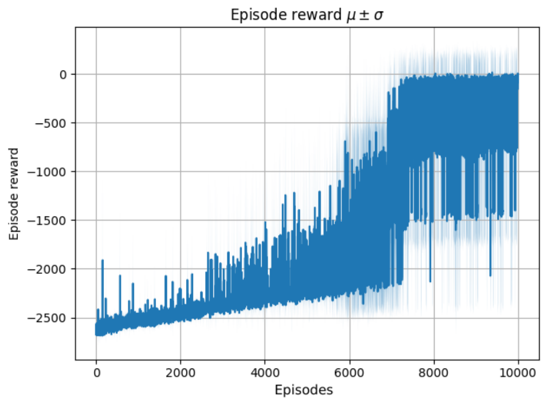
Let's look at the evolution shown in section 4.1.2 and this section compared to section 4.1.1. We can see how the HRL approach suffers a relatively small increase in the execution time, whereas for the Q-learning approach the increase in time is growing linearly with the number of tasks —approximately takes three times longer to learn three tasks than learning just one—. This fact could indicate the benefit of using task specific ER and skill re-utilization of the HRL.

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	4:52 $\pm$ 0:41	$6.77 \times 10^6 \pm 7.8 \times 10^5$	100%	-
Q-learning	1:38 $\pm$ 0:01	$1.39 \times 10^6 \pm 7.8 \times 10^4$	100%	-

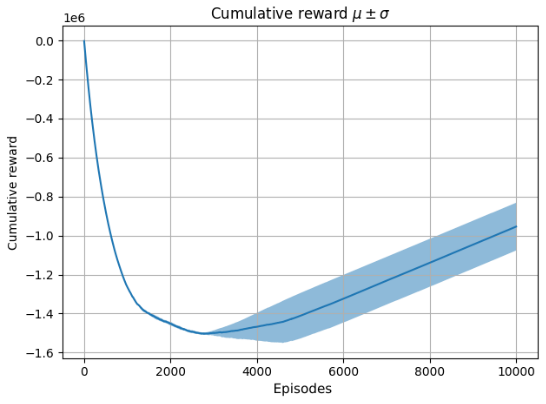
Table 4.3: Learning three tasks. Trained on *Make tea*, *Make pasta* and *Make omelette* tasks



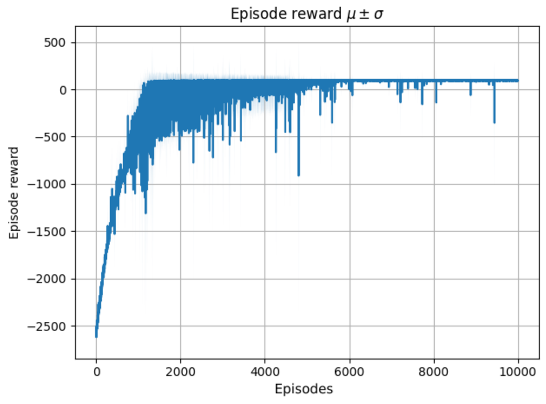
(a) Cumulative reward (HRL)



(b) Episode rewards (HRL)



(c) Cumulative reward (Q-learning)



(d) Episode rewards (Q-learning)

Figure 4.3: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Tasks: *Make tea*, *Make pasta* and *Make omelette*

4.1.4 Learning a New Task (Similar Task)

In this experiment, both approaches are trained on a previously unseen task: *Make soup*. It trains on it after being trained on experiment 4.1.2, so the training order results in:

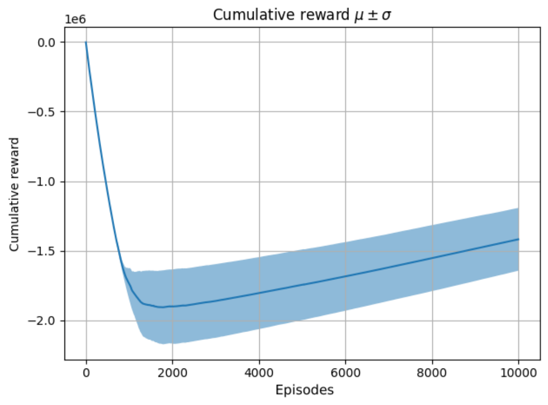
$$(\textit{Make tea} + \textit{Make pasta}) \rightarrow \textit{Make soup}$$

We first make use of Algorithm 3 to train the HRL agent on this new task while trying not to forget previously learned tasks. This new medium-low difficulty task is very similar to the previously seen tasks *Make tea* and *Make pasta*. When comparing the action diagrams (A.2, A.3, A.4) we can see that they share several action sequences. Having a structure so similar helps the system reuse skills and potentially learn faster. The results are shown in Table 4.4 and Figure 4.4.

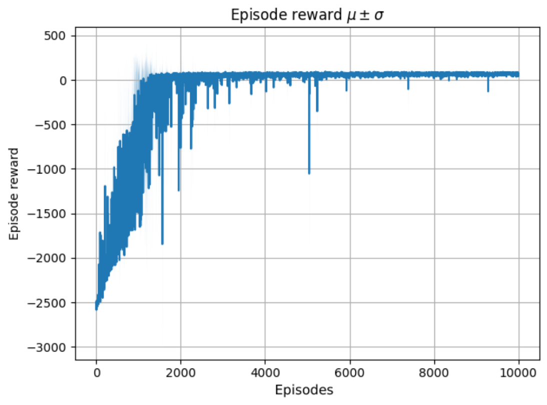
For the first time, we see a significant decrease in the execution time of the HRL

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	1:08 ± 0:13	$1.10 \times 10^6 \pm 2 \times 10^5$	100%	2/2
Q-learning	0:33 ± 0:04	$4.51 \times 10^5 \pm 7.5 \times 10^4$	100%	2/2

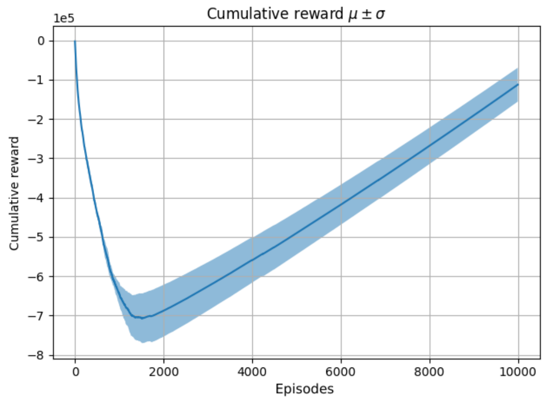
Table 4.4: Learning a new task. Trained on *Make soup* task



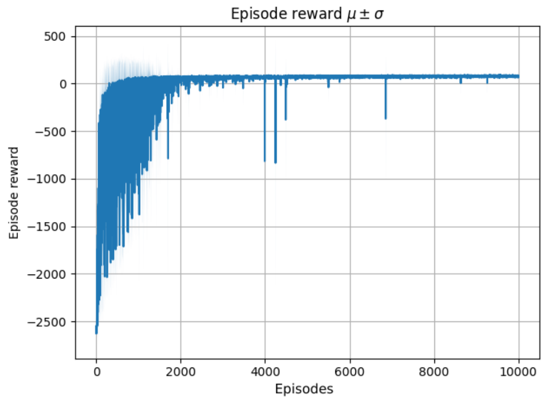
(a) Cumulative reward (HRL)



(b) Episode rewards (HRL)



(c) Cumulative reward (Q-learning)



(d) Episode rewards (Q-learning)

Figure 4.4: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Task: *Make soup*.

approach. Learning a new task from a starting point of knowledge is the context most adequate for the HRL algorithm, and the one we had in mind when designing it. In the process of learning a new task, the master policy can use a combination of the sub-policies at different moments to try to solve the new task. Since the overlap between the tasks is big, only small changes are needed in order to obtain a solution yielded by the combination of one or more sub-policies.

4.1.5 Learning a New Task (Dissimilar Task)

In this experiment, both approaches are trained on a previously unseen new task: *Clean stove*. The agent trains on this new task after experiment 4.1.4, so the training order results in:

$$(\textit{Make tea} + \textit{Make pasta}) \rightarrow \textit{Make soup} \rightarrow \textit{Clean stove}$$

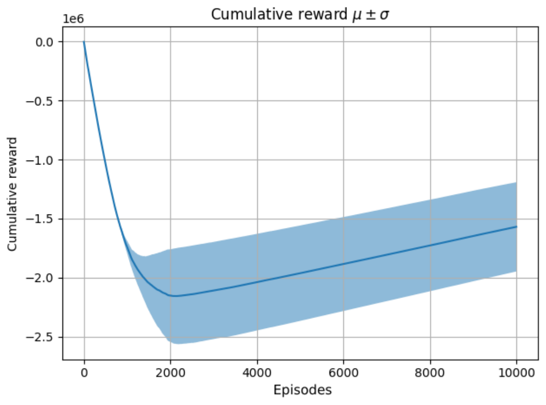
Unlike the previous experiment, the new task has very little in common with the old tasks. Looking at the action diagrams (A.1, A.2, A.4), we can see that *Clean stove* has virtually no resemblance to any of the other tasks. The goal is to observe how the system behaves when it tries to learn a task and it cannot reuse most of the knowledge that it already has. Results are shown in Table 4.5 and Figure 4.5.

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	1:24 ± 0:33	$1.31 \times 10^6 \pm 4.7 \times 10^5$	100%	3/3
Q-learning	0:35 ± 0:15	$4.97 \times 10^5 \pm 2.2 \times 10^5$	100%	3/3

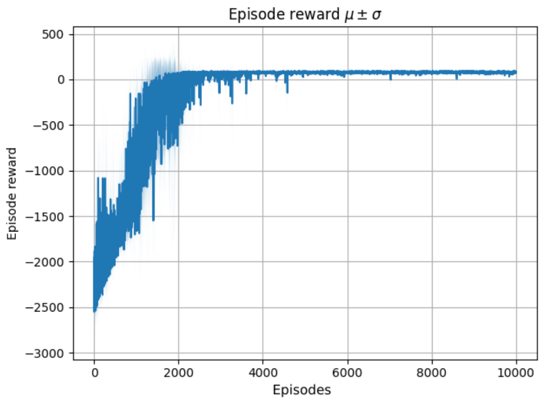
Table 4.5: Learning a new task. Trained on *Clean stove* task

Let's first analyze the standard Q-learning method. The model is able to learn the new task successfully and in a shorter time than its counterpart. However, we can also notice that the execution time is slightly longer than in the previous experiment. Although this new task is significantly easier than the one in experiment 4.1.4, the learning process is slightly longer due to the fact that it needs to collect experience even for the tasks that it has already learnt.

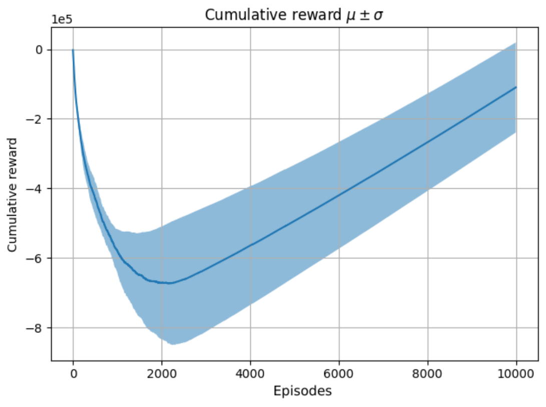
If we look at the HRL approach, we see that it is able to learn the new task. The execution time is slightly longer than the one in the previous experiment. That could be due to the fact that the model needs to perform more updates for previous tasks. The remind block needs to accommodate one more task — the task learnt in the previous experiment —.



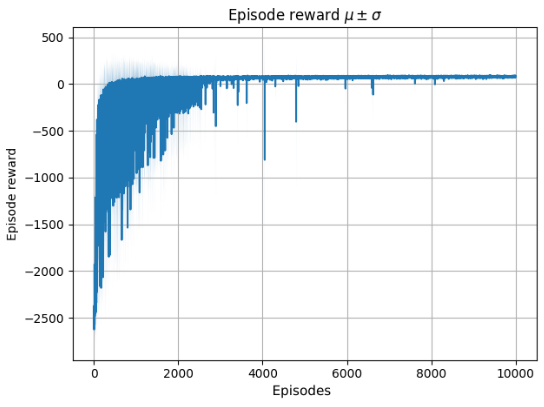
(a) Cumulative reward (HRL)



(b) Episode rewards (HRL)



(c) Cumulative reward (Q-learning)



(d) Episode rewards (Q-learning)

Figure 4.5: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Task: *Clean stove*.

4.1.6 Learning Additional New Task (Similar Task)

In this experiment, both approaches are trained on a previously unseen new task: *Make pancakes*. The experiment is similar to experiment 4.1.4 but with a different starting point. At the start of the training, the system already has knowledge about four tasks. The order in which the system learnt the tasks is:

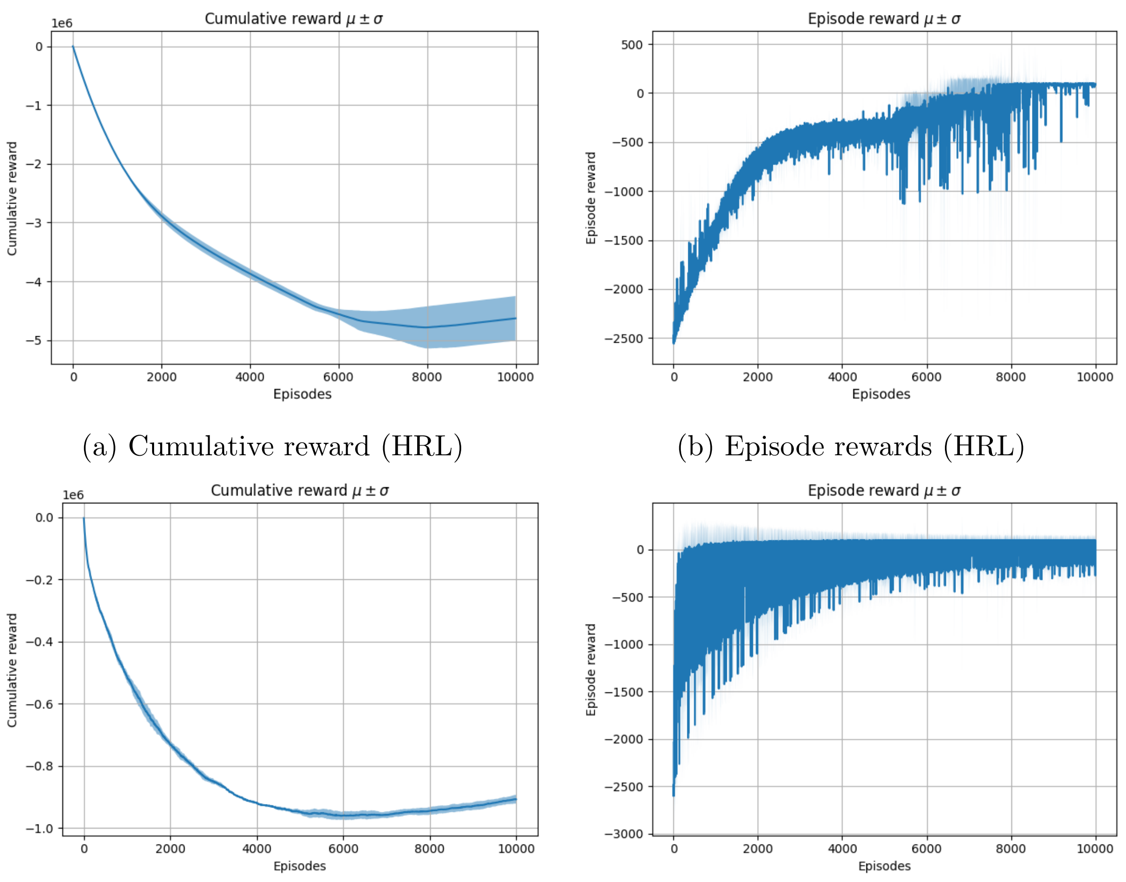
$$(\textit{Make tea} + \textit{Make pasta}) \rightarrow \textit{Make soup} \rightarrow \textit{Clean stove} \rightarrow \textit{Make pancakes}$$

where the first 2 tasks are learnt using Algorithm 2 and the rest were trained sequentially using Algorithm 3.

For the first time, we can observe a significant change in the results of both models. Firstly, the standard Q-learning approach was not able to successfully learn this new task. It might seem when looking at Figures 4.6c and 4.6d that the model was starting to get consistent positive rewards. Nonetheless, the positive rewards

Experiment	Time (hh:mm)	Steps	Accuracy	Retention
HRL	7:18 ± 1:33	$6.58 \times 10^6 \pm 1.2 \times 10^6$	100%	4/4
Q-learning	-	-	0%	4/4

Table 4.6: Learning a new task. Trained on *Make pancakes* task



(a) Cumulative reward (HRL) (b) Episode rewards (HRL)

(c) Cumulative reward (Q-learning) (d) Episode rewards (Q-learning)

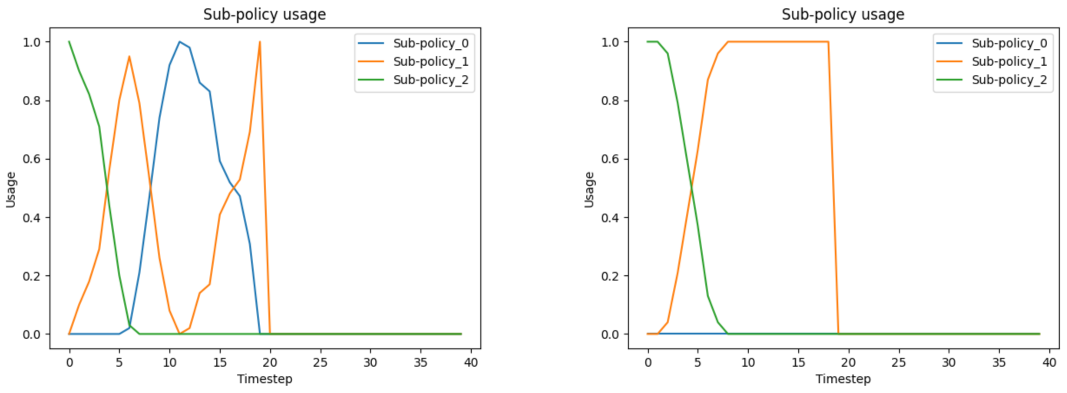
Figure 4.6: Training graphs (cumulative reward and reward per episode) for HRL and basic Q-learning methods. Task: *Make pancakes*.

come from the agent solving old tasks that it had already learnt. In contrast, the HRL approach was able to learn the new task, although it took it a longer time than usual to achieve that.

4.2 Evaluating the Re-usability of Skills for the Proposed Method

As we introduced in previous sections, the main focus of this thesis was to find a system that could effectively and efficiently retain and reuse knowledge. From a pragmatic point of view, this means that if the agent already knows how to boil water, it should be able to reuse that knowledge for other tasks as well instead of learning it again.

In order to verify this hypothesis, the HRL method was tested in the following manner. Once the model has been trained for a specific subset of tasks, one hundred episodes are run over only one of those tasks. In these episodes, the agent always follows its policy ($\epsilon = 0$). During all the runs, the actions of the master policy are recorded — remember that the master policy actions simply indicate the index k of which sub-policy to follow in that timestep —. Once the 100 episodes conclude, we calculate the number of times that each policy was used at each timestep. This gives an idea of how policies were used. For example, ϕ_1 was used at the beginning of the task but then ϕ_3 was used to finish it.



(a) Test run using model obtained after experiment 4.1.2; Chosen task: *Make tea*

(b) Test run using model obtained after experiment 4.1.4; Chosen task: *Make soup*

Figure 4.7: Sub-policy selection

Figure 4.7 shows the probability of choosing a specific sub-policy in any given timestep. If we look at the green line (ϕ_2), we can see that it is predominantly used at the beginning of the episode for both tasks. This could indicate that the agent uses the same sub-policy to solve the first steps in both situations. If we look at the action diagrams A.2 and A.3, we can observe that both action sequences are very similar. For instance, those first steps using sub-policy ϕ_2 could represent the agent approaching the cabinet, opening the door, getting the ingredient and closing the door.

After those first steps, Figure 4.7a uses a combination of ϕ_0 and ϕ_1, while Figure 4.7b uses mainly ϕ_1. This behaviour is completely understandable. After all, it is perfectly plausible that a single sub-policy can contain the knowledge on solving whole tasks. This behaviour is shown in Figure 4.8. In this case, both experiments

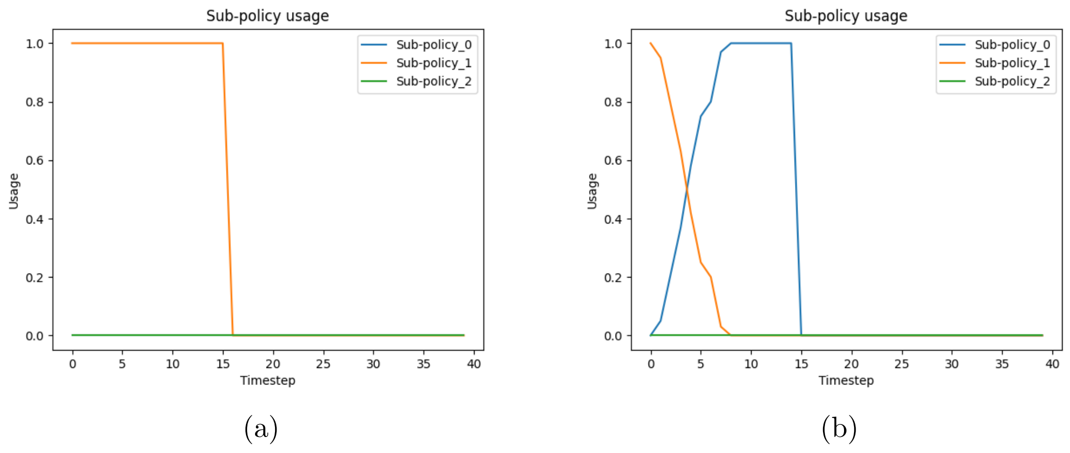


Figure 4.8: Sub-policy selection; Test run using model obtained after experiment 4.1.5; Chosen task: *Clean stove*

are run using the same model and focusing the same task. However, the master policy is re-trained from scratch before starting the measurements. While Figure 4.8b shows a combination of sub-policies ϕ_0 and ϕ_1, Figure 4.8a only shows the usage of sub-policy ϕ_1. In this case, sub-policy ϕ_1 is capable of solving the proposed task without the need of any other sub-policies. Depending on the state of the master policy after training, this task can be solved by using a combination of different sub-policies or by constantly using the same sub-policy.

Chapter 5

Discussion

We conclude the final chapter of this thesis with a discussion. First, we give answers to the original research questions presented in Section 1.1. After that, we suggest possible directions for future work. Lastly, there is a brief section to summarize the efforts done in this thesis and the results.

5.1 Answers to Research Questions

Can we design a system that can learn new tasks throughout its lifetime and effectively and efficiently retains and reuses previously gained knowledge to solve new problems?

The system designed in Section 3.2 is indeed capable of learning new tasks sequentially, and it is able to effectively and efficiently retain and reuse previously gained knowledge to solve new problems. The proposed model was able to learn all the tasks in all the experiments. Moreover, the process of learning a new task was faster when the model was using previously gained knowledge. This indicates the efficient reuse of previous knowledge by the system. Furthermore, we can say that the system effectively retains previously gained knowledge since it never forgot previously seen tasks.

How does this system perform in a specific scenario where a simulated household robot needs to solve common tasks?

The system was tested in such a scenario and the results were majorly positive. Any potential household robot that will be developed in the future needs to be able to learn new tasks according to the user's needs. Any system that can deal with this constraint is well suited for this situation. After all the experiments conducted in Section 4.1, we have proven that the proposed system can continuously learn new tasks without forgetting the old ones. Moreover, it uses past knowledge to accelerate the learning speed of new related tasks.

How does this system perform compared to a standard reinforcement learning algorithm such as Q-learning?

Although the standard Q-learning algorithm required some minor modifications to make it suitable for a multi-task LML approach, it performed slightly better than the proposed model in terms of learning speed. The only unsuccessful experiment was 4.1.6. The problem could be due to the saturation of the network. Making the MLP larger could fix the issue, but it would also require a complete re-training of the model. The HRL approach took significantly longer than usual in said experiment, which could also point to the need of expanding the network. However, expanding the model in the HRL is as simple as adding a new MLP representing a new sub-policy. The HRL has the advantage of being more modular which makes it more suitable for an LML approach.

Is such a system a suitable option for a lifelong learning scenario, such as the household robot environment?

As mentioned earlier, a household robot needs to be able to learn new tasks as it is required. The characteristics of the proposed method make it very suitable for lifelong learning scenarios. The model is able to remember previously learnt tasks and it has the ability to use previous knowledge to speed up the learning process of new tasks. The system can be expanded by adding new sub-policies to the system with no re-training of the old policies required.

5.2 Future Work

The system proposed in this thesis proved to be a viable solution for LML in scenarios similar to the one used. Although it underperformed in terms of learning speed when compared to a more standard Q-learning model, it was able to solve experiment 4.1.6 while its counterpart could not. The results obtained in the experiments conducted show that this new system has the potential to become a system that can be used in real-life agents. Nevertheless, more work would be needed to make that thought a reality. In this section we present several areas where more research could drastically improve the quality of this approach.

One direction for future work could be to expand the task set. In this thesis we limited the scope of the tasks to the kitchen domain. Most of them shared actions but still all of them belonged to the same domain. It would be very interesting to see how the system behaves when there are several sets of similar tasks. For instance, a set of living room tasks — such as mop the floor, clean up the table, clean the windows, etc. — and another set of bedroom tasks — such as make the bed, fold clothes, vacuum the carpet, etc.—. Having more diversity in the task distribution could potentially indicate flaws in the system that are now undetectable. It could also indicate directions for improvement. Additionally, different ways of presenting the goal task to the agent could be tested. For example, trying to find ways to encode tasks based on its action sequence. Just as word embeddings are created using semantics for NLP approaches, tasks could be encoded based on the actions that are required to solve them. This way, the vector encoding for making tea would be much closer to the vector of making soup than to the vector of cleaning stove.

As more tasks are added, there will come the time where the system needs to be expanded. The system was designed with LML in mind, so expanding the model is a simple process. It is possible to add new sub-policies as needed. In theory, the system should be able to use this new sub-policies to solve new tasks, while old policies remain constant and are able to solve older tasks. This behaviour occurs because the master policy decides which sub-policies to follow/train depending on the task at hand. More work in this area is needed in order to check the limits of this hypothesis.

Perhaps, the most relevant improvements to this approach could come from increasing the learning speed of the algorithm. As it was shown in the experiments, the proposed model underperformed in terms of speed when compared to its counterpart. This issue was partly due to the fact that the master policy is re-trained in every cycle. The rationale for resetting the master policy θ was inspired by the work of Frans et al. (2017). *"As we update the sub-policy parameters ϕ while reusing master policy parameters θ, we are assuming that re-training θ will result in roughly the same master policy. However, as ϕ changes, this assumption holds less weight."* This is the reason why we re-train θ once a threshold of I_ϕ iterations has passed. Finding ways to overcome the need for constant re-training of θ could eventually yield a faster algorithm. We actually performed some initial experiments and, for example, experiment 4.1.4 saw its learning time cut by more than half when we stopped re-initializing θ.

Another direction for future work would be to experiment with different kinds of experience replay. Its been shown that using more advanced types of experience replay such as Hindsight Experience Replay (Andrychowicz et al., 2017) or Prioritized Experience Replay (Schaul, Quan, Antonoglou, & Silver, 2015) can increase the learning speed and even make unsolvable tasks solvable. Using other types of ER could make the proposed algorithm faster and more robust.

5.3 Conclusion

In this thesis we developed a hierarchical reinforcement learning system that can be used in a lifelong learning scenario. Such a scenario is represented as a simulated household robot that is able to learn new tasks presented by the user. The agent is able to learn new tasks throughout its lifetime while retaining the knowledge on how to solve old tasks. Moreover, the agent re-uses previously acquired knowledge to solve new tasks. The efficient transfer of knowledge from one task to another makes this system suitable for lifelong machine learning. Additionally, the architecture of this system allows it to be expanded as the situation requires, with very little re-training needed (only master policy needs re-training). This modularity and ease to expand the size of the model is another key factor for an LML system.

In conclusion, the proposed system is an effort to bring us a step closer towards the future of LML. Systems such as household robots, robots sent on exploratory missions in space or under the sea could benefit from advancements in this area. We hope that this thesis inspires other researchers so that we can someday achieve the dream of real autonomous robots.

Bibliography

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., ... Zaremba, W. (2017). Hindsight Experience Replay. *Advances in neural information processing systems*, *30*, 5048–5058.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, *13*(1-2), 41–77.

Bellman, R. E. (1961). *Adaptive control processes: A guided tour.* Princeton University Press.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. eprint: arXiv:1606.01540

Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in neural information processing systems* (pp. 271–278).

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of artificial intelligence research*, *13*, 227–303.

Frans, K., Ho, J., Chen, X., Abbeel, P., & Schulman, J. (2017). Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767.*

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* http://www.deeplearningbook.org. MIT Press.

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with HEXQ. In *ICML* (Vol. 19, pp. 243–250).

Hengst, B. (2012). Hierarchical approaches. In *Reinforcement learning: State-of-the-art* (pp. 293–323). doi:10.1007/978-3-642-27645-3_9

Hutter, M. (2007). Universal algorithmic intelligence: A mathematical top-down approach. In *Artificial general intelligence* (pp. 227–290). Springer.

Kerzel, M., Mohammadi, H. B., Zamani, M. A., & Wermter, S. (2018). Accelerating deep continuous reinforcement learning through task simplification. In *2018 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–6). IEEE.

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274.

Levy, A., Platt, R., & Saenko, K. (2018). Hierarchical reinforcement learning with hindsight. *arXiv preprint arXiv:1805.08180.*

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*(3-4), 293–321.

Luo, S., Kasaei, H., & Schomaker, L. (2020). Accelerating reinforcement learning for reaching using continuous curriculum learning. arXiv: 2002.02697 `[cs.AI]`

McCarthy, J. (1998). What is artificial intelligence? Retrieved from http://cogprints.org/412/

McClelland, J. L., Rumelhart, D. E. et al. (1986). Parallel distributed processing. *Explorations in the Microstructure of Cognition*, *2*, 216–271.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. doi:10.1038/nature14236

Moerman, W., Bakker, B., & Wiering, M. (2007). Hierarchical assignment of behaviours to subpolicies. NIPS* 2007 workshop on Hierarchical Organization of Behavior: Computational ...

Nachum, O., Gu, S. S., Lee, H., & Levine, S. (2018). Data-efficient hierarchical reinforcement learning. In *Advances in neural information processing systems* (pp. 3303–3313).

Niel, R., & Wiering, M. A. (2018). Hierarchical reinforcement learning for playing a dynamic dungeon crawler game. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 1159–1166). IEEE. doi:10.1109/SSCI.2018.8628914

Rosenblatt, F. (1962). *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Spartan Books, Washington DC.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation*. California Univ San Diego La Jolla Inst for Cognitive Science.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. doi:10.1038/323533a0

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv: 1511.05952 [cs.LG]

Scikit-learn. (2019). Multi-layer perceptron. Retrieved June 28, 2020, from https://scikit-learn.org/stable/modules/neural_networks_supervised.html

Silver, D. L., Yang, Q., & Li, L. (2013). Lifelong machine learning systems: Beyond learning algorithms. In *2013 AAAI spring symposium series*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... Lanctot, M., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Bolton, A., et al. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. Cambridge, MA, USA: A Bradford Book.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, *112*(1-2), 181–211.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, *6*(2), 215–219.

Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J., & Mannor, S. (2016). A deep hierarchical approach to lifelong learning in Minecraft. *arXiv preprint arXiv:1604.07255*.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation, University of Cambridge).

Watters, N., Matthey, L., Bosnjak, M., Burgess, C. P., & Lerchner, A. (2019). COBRA: Data-Efficient Model-Based RL through Unsupervised Object Discovery and Curiosity-Driven Exploration. arXiv: 1905.09275 [cs.LG]

Wiering, M., & Schmidhuber, J. (1997). HQ-learning. *Adaptive Behavior*, *6*(2), 219–246.

Appendix A

Action Diagrams

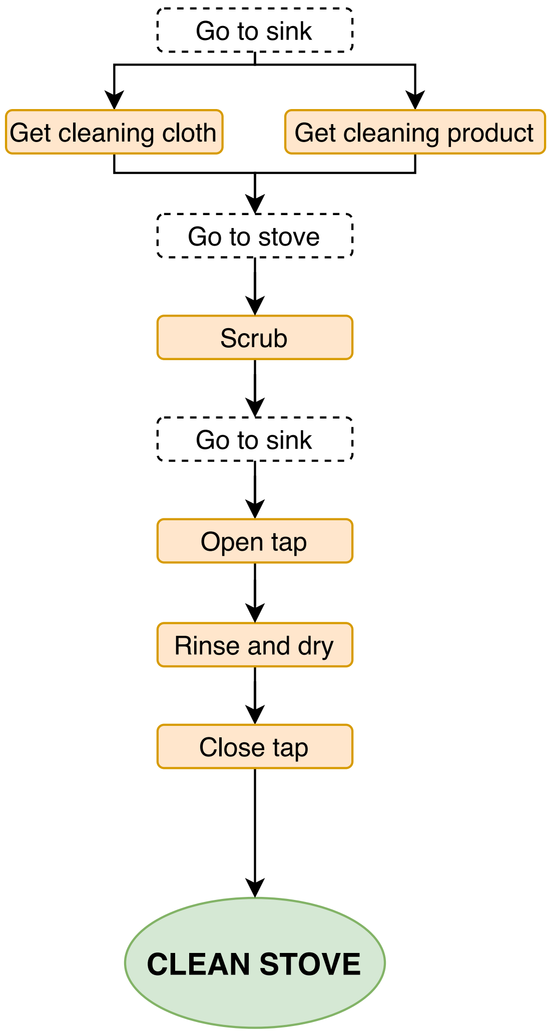


Figure A.1: Clean stove action diagram

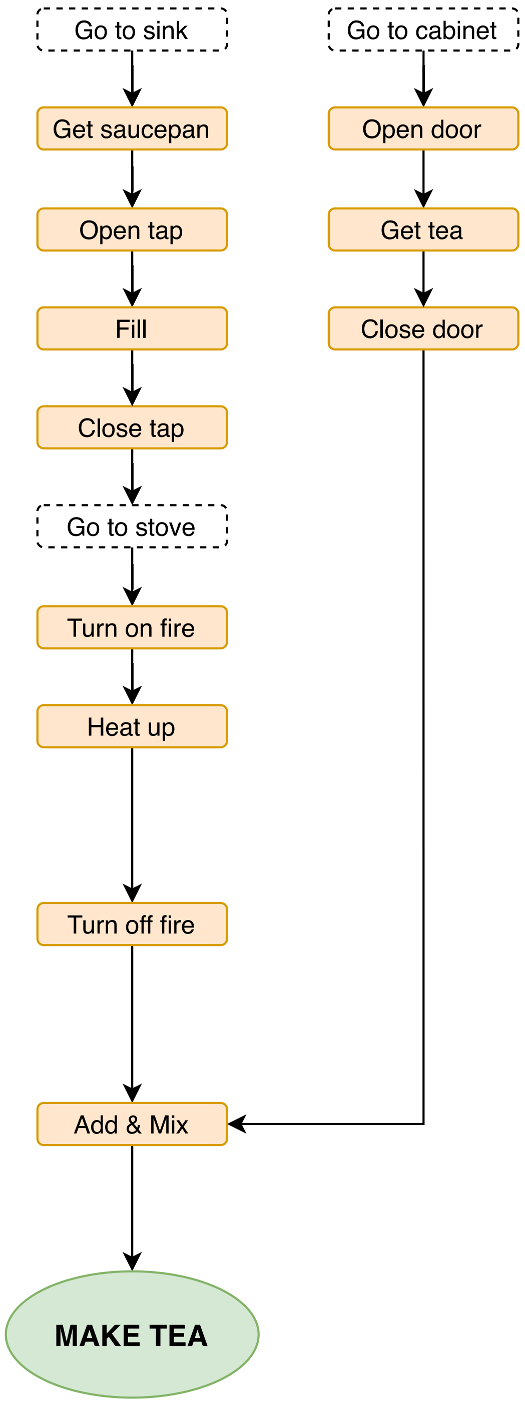


Figure A.2: Make tea action diagram

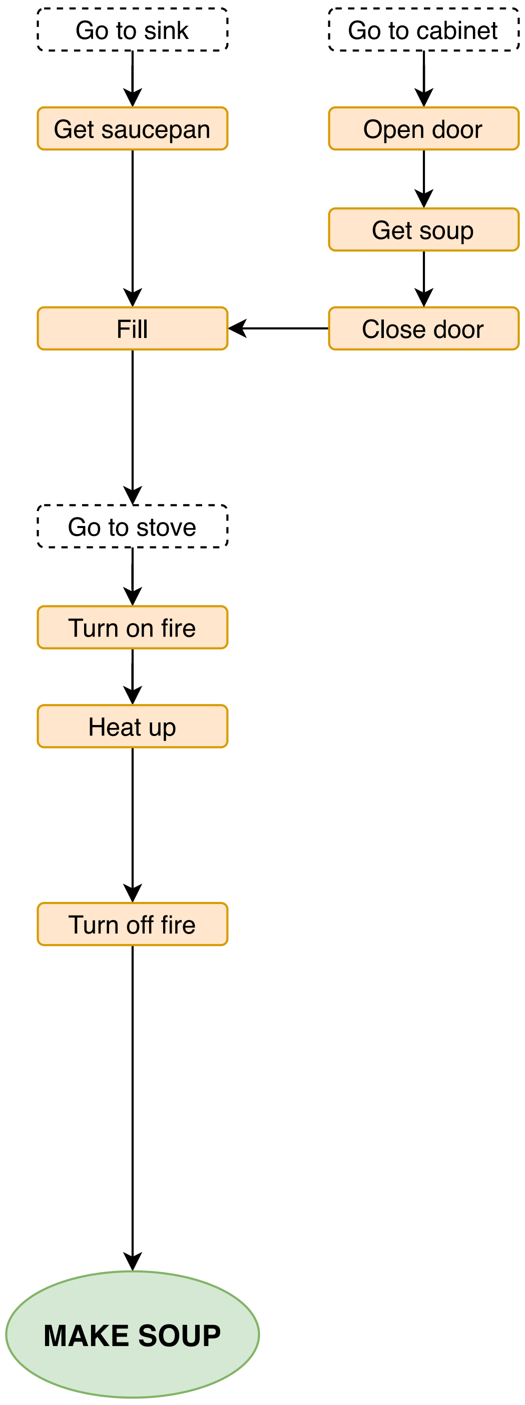


Figure A.3: Make soup action diagram

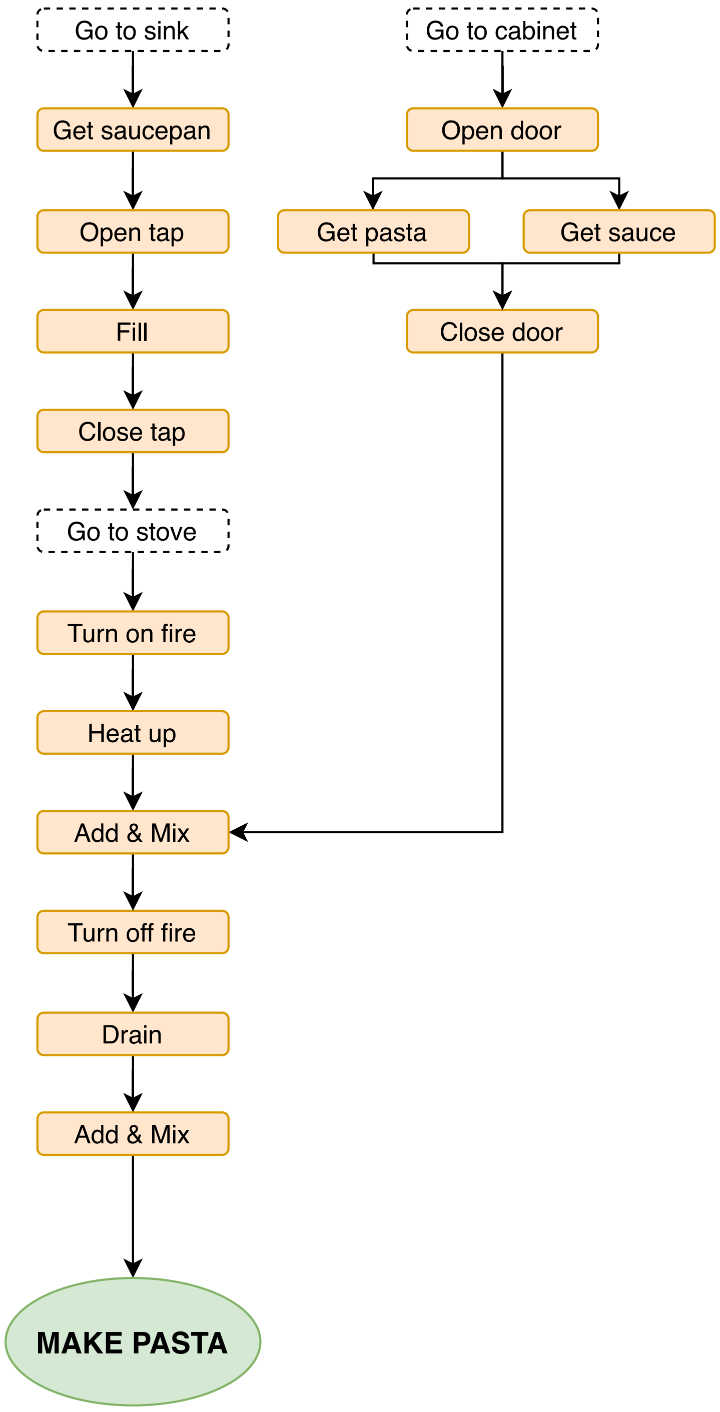


Figure A.4: Make pasta action diagram

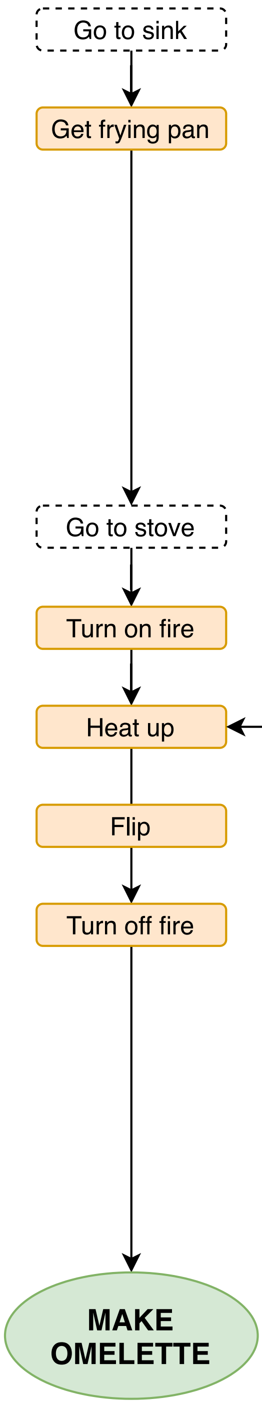


Figure A.5: Make omelette action diagram

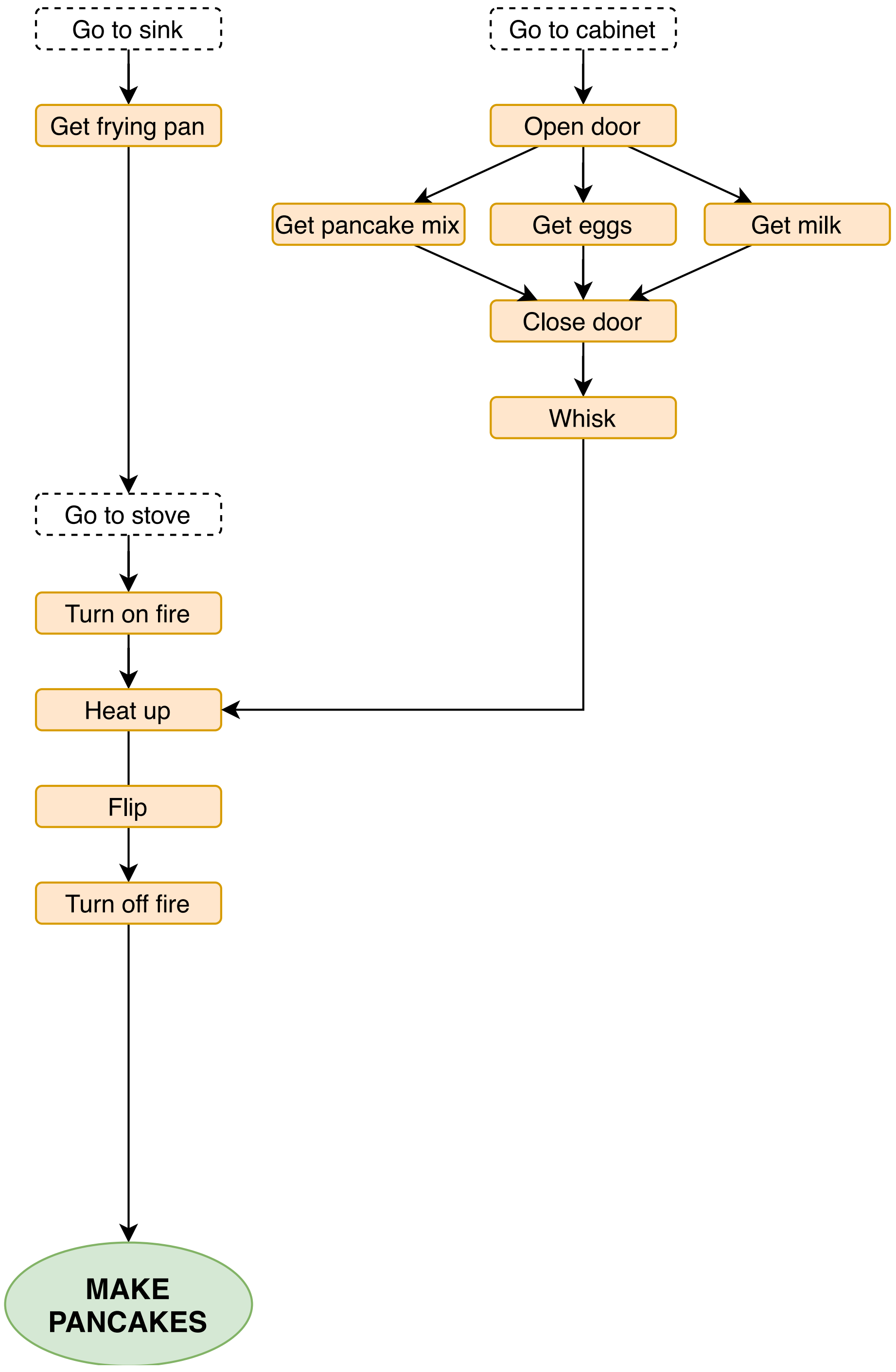


Figure A.6: Make pancakes action diagram

Appendix B

Hyperparameters

C (number of training cycles)	1×10^4
N (collect experience timesteps)	2000
R (remind frequency)	10
ρ (success threshold)	75%
Batch size	32
I_θ	500
I_ϕ	500
$\epsilon_{start}\,(\theta)$	1
$\epsilon_{end}\,(\theta)$	0.1
$\epsilon_{decay}\,(\theta)$	450
$\epsilon_{start}\,(\phi)$	1
$\epsilon_{end}\,(\phi)$	0.1
$\epsilon_{decay}\,(\phi)$ (Alg. 2)	1×10^7
$\epsilon_{decay}\,(\phi)$ (Alg. 3)	5×10^5
$\gamma\,(\theta)$	0.9
$\gamma\,(\phi)$	0.9
$\alpha\,(\theta)$	1×10^{-2}
$\alpha\,(\phi)$	2.5×10^{-4}
ER_θ	2000
ER_ϕ	1×10^5
Q-Net target update (θ)	20
Q-Net target update (ϕ)	1000
MLP (θ)	2 layers (1000/500)
MLPs (ϕ_k)	2 layers (1000/500)

Table B.1: Hyperparameters used for experiments with HRL (Section 4)

C (number of training episodes)	1×10^4
ρ (success threshold)	75%
Batch size	32
$\epsilon_{start}\,(\theta)$	1
$\epsilon_{end}\,(\theta)$	0.1
$\epsilon_{decay}\,(\theta)$	5×10^5
γ	0.85
α	2.5×10^{-4}
Q-Net target update	1000
ER	1×10^6
MLP	2 layers (1000/500)

Table B.2: Hyperparameters used for experiments with Q-learning (Section 4)