OPEN-ENDED CONTINUOUS REINFORCEMENT LEARNING FOR MOBILE ROBOTS

A Dissertation Presented

by

Paresh Dhakan

Submitted for the degree of

Doctor Of Philosophy

to

Ulster University

Faculty of Computing, Engineering and the Built Environment

May 2022

ABSTRACT

Creating an intelligent agent capable of open-ended learning and long term autonomy is still an active research area. Reinforcement learning, where an agent learns by interacting with its environment, is suitable for agent autonomy and its extensions, such as motivated reinforcement learning and goal-oriented agent architectures, with their focus on meta-cognitive aspects such as 'what to learn', enable autonomous multitask learning. The other essential aspect is the cognitive aspect of 'how to learn', a focus area of lifelong learning architectures. When these aspects are combined, it creates a comprehensive agent architecture that would endow an agent to learn new skills with minimal human intervention. The first contribution of this thesis is an agent architecture consisting of a task generation module, knowledge management module and learning module, providing the agent with open-ended, continuous and autonomous learning capabilities. Further, this thesis contributes to each of the modules of this architecture as follows.

In reinforcement learning, the agent learns by interacting with its environment, guided by reward. However, for many dynamic environments, it is unknown upfront what tasks the agent will need to learn, and research has acknowledged the benefits of generating subtasks to direct learning. Depending on how the broken-down subtasks are considered to be accomplished, they can be an achievement, maintenance, approach or avoidance type. The second contribution of this thesis, related to the learning module of the proposed architecture, is a reward design based on these different types of tasks. For a continuously learning agent, tasks direct what the agent learns. Typically, the task design requires external intervention, thus hindering the agent's autonomy. The third contribution of this thesis, related to the task generation module of the proposed architecture, is a mechanism to generate tasks at different levels of complexity. That enables the agent to learn simpler, more primitive tasks first, followed by more difficult compound tasks.

Furthermore, one of the key characteristics of continuous learning is that the agent should be able to use its existing knowledge to solve future tasks. Compound tasks can be either a sequential or a concurrent combination of primitive tasks. The fourth contribution of this thesis, related to the knowledge management module of the proposed architecture, is a compositionality technique whereby the agent can combine its primitive skills for disjoint tasks to solve a compound task that is a concurrent combination of those tasks. Finally, the fifth contribution of this thesis is metrics to measure task difficulty, agent's competency for a particular skill and agent performance for tasks of different types. A mobile robot is used for all the experiments to show how the agent generates new tasks, learns solutions to those tasks and combines the skills to accomplish compound tasks, thus demonstrating autonomous behaviour of continuous learning in an open-ended way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

AC	Actor-Critic
ART	Adaptive Resonance Theory
BDI	Belief Desire Intention
IMOL	Intrinsically Motivated Open-Ended Learning
IMRL	Intrinsically Motivated Reinforcement Learning
KL divergence	Kullback-Leibler divergence
MDP	Markov Decision Process
MFRL	Motivated Flat Reinforcement Learning
MIRL	Motivated Introspective Reinforcement Learning
MMORL	Motivated Multi-Option Reinforcement Learning
NN	Neural Network
RL	Reinforcement Learning
SART	Simplified Adaptive Resonance Theory
TD	Temporal Difference

PUBLICATIONS FROM THIS THESIS

1. ***P. Dhakan**, K. E. Merrick, I. Rano, and N. Siddique, "Modular Continuous Learning Framework," 2018 Joint IEEE 8th International Conference on Development and Learning and Epigenetic Robotics, ICDL-EpiRob 2018. Tokyo, Japan, pp. 107–112, 2018.*
    - *This forms part of Chapter 3 of this thesis (Agent Architecture For Open-Ended and Continuous Learning).*

2. ***P. Dhakan**, K. E. Merrick, I. Rano, and N. Siddique, "Intrinsic rewards for maintenance, approach, avoidance, and achievement goal types," Frontiers in Neurorobotics, vol. 12, no. October. 2018.*
    - *This forms part of Chapter 4 of this thesis (Reward Design for Autonomous Learning).*

3. ***P. Dhakan**, K. Kasmarik, I. Rano, and N. Siddique, "Open-Ended Continuous Learning of Compound Goals," IEEE Transactions on Cognitive and Developmental Systems, vol. 13, no. 2. pp. 274–285, 2019.*
    - *This forms part of Chapter 5 of this thesis (Self-Generation of Tasks to Direct the Learning).*

4. ***P. Dhakan**, K. Kasmarik, P. Vance, I. Rano, and N. Siddique, "Concurrent Skill Composition using Ensemble of Primitive Skills.". IEEE Transactions on Cognitive and Developmental Systems, Accepted for publication – 10th May 2022.*
    - *This forms part of Chapter 6 of this thesis (Reuse of Learned Knowledge by Skill Composition).*

5. *N. Siddique, **P. Dhakan**, I. Rano, and K. E. Merrick, "A review of the relationship between novelty, intrinsic motivation and reinforcement learning," Paladyn, Journal of Behavioral Robotics, vol. 8, no. 1. pp. 58–69, 2017.*

CHAPTER 1 INTRODUCTION

1.1 Introduction

Designing an intelligent agent is complex [1] [2] [3]. In the case of living organisms, while some skills are self-learned, others need to be taught. While some learning is structured, most of it is unstructured. Moreover, the reason to attempt to learn some skills ranges from survival to social and sometimes just for its sake [4]. In any case, for a living organism, there appears to be an ecosystem that initiates, drives and sustains learning. To create an artificial agent, all of this has to be thought of and designed [5] [6] [7] [8]. The reason, the drive, the motivation, and the cause to learn something new is all a piece of code or some form of a programmable circuit. The knowledge representation, in its simple form, is a relationship mapping between data points. The knowledge repository, in its simplest form, is a data store. In the 1980s and 1990s, this used to be a very active research area [9] [10]. Since then, several tangible and intangible advances have been made in everything even remotely related to the intelligent agent, from hardware (processing power, sensor technologies, to name a few) and software (learning frameworks, knowledge representation, to name a few) to our understanding of what intelligence is [3]. In recent years researchers have started to examine self-motivated, continuous learning agents [11] [12] [13]. However, there remain few established architectures that empower the agents to self-direct their learning. That is the topic of this thesis.

An agent or a robot, in most cases, is designed to do a particular task or a set of tasks. Undoubtedly, those repetitive tasks are carried out with high precision and better than humans in many cases. However, due to one failure in hardware or software or one change to the environment, such an agent's behaviour becomes highly unpredictable. Even if the agent has the capability to adapt to programmed predictable uncertainty, such as sudden road closure requiring the mapping software to recalculate the route, in many cases, any unexpected change in the normal working condition leads to failure. Such adaptability requires the agent to learn continuously, sort of rewiring some of the internal representations of the model of its world on a constant basis, and this should continue for the whole lifetime of the agent. That is commonly referred to as lifelong or continual learning [14] [15]. That learning behaviour is reactive since the learning is triggered by an event [2]. On the other hand, a proactive learning behaviour can be where the agent, like living organisms, learns a skill to prevent some mishap or find a more optimal solution without being asked to do that [2]. That requires the agent to be able to create a plan, a sort of curriculum of what it should learn next, or do next, given its current level of knowledge. Such structured learning is referred to as incremental or curriculum learning [16] [17]. When the agent does this with no immediate benefit to itself, it is called motivated learning [18] and the behaviour exhibited by the agent is said to be open-ended learning [4]. Open-ended learning enables the agent to self-direct its learning based on its interaction with the environment. With its focus on the meta-cognitive aspects such as what to learn [19] and when to learn [20], it is envisaged to empower the agent such that constant supervision will no longer be necessary. When intrinsic motivation [4], in particular, curiosity, is used to direct what the agent learns, it creates an intrinsically motivated open-ended learning framework [12] [21]. Consider a service robot in a real-world situation. Since the environment is dynamically changing, it is not possible to contemplate all the skills that the service robot will require [22]. Since it is not practical to provide the agent with constant guidance regarding which environment-specific skills to learn, it would be essential that the agent can decide by itself which skills to acquire, i.e. learn in an open-ended manner.

Closely related to the above is lifelong learning, whose general characteristics are that it learns continuously, accumulates and incorporates new learning into its knowledge repository, and can reuse the learned knowledge to find solutions to future tasks [15] [23]. That enables the agent to continuously increase the overall knowledge of its environment. From the literature review, it appears that the questions that lifelong learning is aiming to find answers to are: (i) how to overcome the "Frame Problem", i.e. if the system is limited by what it models, is it possible to make the same system learn other things as well [24], (ii) if all the knowledge is stored in a single representation, how does one overcome the problem of catastrophic forgetting when a new task is to be learned [25], (iii) what is a representation of the core commonality of the related tasks and how best to transfer known skill to a related new task, and (iv) learning how to learn [23] [26]. That shows that this research area's focus is on the cognitive aspect.

For a comprehensive agent architecture, both open-ended learning, with its focus on the meta-cognitive aspect, and lifelong learning, with its focus on the cognitive aspect, are required [27]. The third aspect of all this is the agent's ability to learn independently or with minimal intervention from other agents/humans [28]. Most real-world situations cannot easily be divided into training and test samples, so the agent should be able to gather its data and make a deliberate attempt to learn aspects of the skill it is uncertain about. Since the learning rate of different skills is not going to be the same, the agent should be able to tune its learning parameters. Autonomy, defined as the system's ability to make its own decisions, is required for the agent to be classified as a functional artificial agent [28]. Reinforcement learning is a learning method where the agent interacts with its environment and learns new skills by trial and error. The agent is not instructed what action it should take in a particular state of its world but must figure out by itself the best action it should take from all the available actions. Over time, it forms a policy, i.e., mapping between states and actions akin to skill. A policy can be packaged and stored. It can also be used as a macro-action and can be recalled based on a trigger. It executes the packaged behaviour and terminates when a final state is reached. These attributes have made reinforcement learning an ideal starting point for lifelong learning architectures [29] [30] [31]. However, as the literature review shows, there is no agreement on the necessary components to take reinforcement learning from reliable task-oriented learning to reliable lifelong open-ended learning.

With that gap as the starting point, Section 1.2 lists the research questions. The solutions to those questions, i.e. the contributions of this research, are summarised in Section 1.3. Then, Section 1.4 details the organisation of this thesis.

1.2 Research Questions

This research starts with the premise that the essential aspects of the architecture of a functional self-learning agent are that it should be able to learn in an open-ended manner throughout its whole life and use reinforcement learning to carry out the learning with minimal external intervention. With that in mind, the following are the questions that this research will aim to contribute towards.

Question 1: What are the modules of an open-ended and continuous reinforcement learning architecture?

By the end of Chapter 2, this thesis identifies that an open-ended architecture: i) should be able to decide what new skill it should acquire, ii) should have a memory module where the skills are stored, and iii) the agent should be able to assimilate the knowledge acquired and use that knowledge to solve future tasks. The agent architecture should be modular in that it should be able to add/remove/plugin the auxiliary modules required. The scope of this thesis is limited to reinforcement learning as the learning method. The learning, however, should be domain-independent and hence flexible in the choice of task generation technique and knowledge store technology. With these points in mind, the subsequent research questions focus on the design of specific modules that will sit within an open-ended, continuous reinforcement learning architecture.

Question 2: How does one design a module to generate task-independent reward functions for different types of tasks, including when the primitive tasks are combined to form a compound task?

In reinforcement learning, the learning is guided by feedback, commonly referred to as a reward. How well the agent learns is often determined by how well the reward function is designed. Typically, the reward design is task-dependent and requires significant domain knowledge. That, however, limits the autonomy of the agent. A review of the literature in the area of task-independent reward function design shows that a concept called 'intrinsic motivation' can be used. The agent generates such a reward based on its perceived novelty of the task or an internal prediction error. As an alternative, one could exploit the inherent property of the task type, which would make the reward task-independent. Tasks can be classified as achievement, maintenance, avoidance and approach type [32]. This question focuses on the design of reward functions based on such task classification. Also, one may wonder could such a reward design be extended to be used when the primitive tasks are combined to form a compound task.

Question 3: How does one design a module to self-generate tasks of varying complexity?

To exhibit open-ended learning, the agent should be able to self-generate the tasks to direct what it learns. Also, since the architecture allows continuous learning and requires minimal external intervention, the architecture should support agents to progressively build their knowledge of their environment. For that, the agent must be able to generate tasks of varying complexity, not necessarily hierarchical. It will initially start by learning to attain simpler tasks and gradually, as it gains more knowledge to attain complex tasks [22]. Existing task generation techniques either generate flat tasks or hierarchical tasks. Often tasks are a combination of primitive tasks. For example, consider a task for a robot to open the lid of a bottle. It is a complex task that comprises grasping the bottle and opening the lid. It is a combination of tasks that are not hierarchical. A literature review in the area of self-task generation shows that technique to generate tasks of varying complexity does not exist. Such a technique would enable the agent to start with little or no knowledge of its environment and improve its capabilities over time. Also, it will enable the agent to reuse its skills for primitive tasks to learn compound tasks.

Question 4: How does one design a module to compose a skill for a compound task by combining primitive skills?

The reinforcement learning community's focus has mostly been on devising algorithms to enable faster learning; however, reinforcement learning is sample inefficient by nature. That is because the agent has to try out all the actions when in a particular state to be able to build an accurate model of its environment. Techniques such as imitation learning are employed to make learning more sample efficient [33]. Since the agent architecture has the capability to learn continuously and assimilate the learned knowledge into its repository, an alternative approach to enable faster learning is to reuse the learned knowledge. For related tasks, the knowledge learned from previous tasks can be used [34]; however, not always are the tasks related. Again, consider the example of a robot learning to open the lid of a bottle. The task of grasping is not necessarily related to the task of opening the lid. One approach to learning such compound tasks is using multitask reinforcement learning [35], where the agent can learn multiple/complex tasks from scratch, which, however, is not "sample efficient". As an alternative approach, compositionality can be used, where the skills of primitive tasks are combined to form solutions for compound tasks. In reinforcement learning, this typically has been used when the skills, i.e. policies, are represented as Q-tables. However, such policy representation does not scale to problems with high dimensional or large state spaces. Another way to represent a policy is by using a neural network. In the last few years, the composition of skills represented by the neural network has been an active area of research. Such composition serves as an alternative to 'sample efficient reinforcement learning' and forms a repository of primitive skills that can be mixed and matched to create a variety of sophisticated skills.

1.3 Contributions and Significance

This research makes the following contributions to fulfil the questions listed in the previous section.

1.3.1 A modular agent architecture for open-ended continuous reinforcement learning

This research proposes a modular learning architecture detailed in Chapter 3, Section 3.3. It comprises (i) a task generation module fulfilling the criteria of making the architecture capable of open-ended learning, (ii) a knowledge repository that stores learned skills, fulfilling the criteria of making the architecture capable of continuous learning, and (iii) a learning module which can be any reinforcement learning algorithm. The architecture is flexible and allows any self-task generation technique to be used and any form of knowledge store to be used. The skills can be stored as individual policies, making them easy to recall and combine as required. Using simulated e-puck mobile robot based experiments, Section 3.4 shows how the robot experiences its environment, self-generates its task, learns the skills, and continues that throughout its lifetime. Thus gradually improving its capabilities in an open-ended and continuous manner.

1.3.2 Task-independent reward functions based on the type of task

Tasks can range in abstraction from high-level to low-level. Another categorisation is based on the functional aspect and how it is considered to be attained. The common categories are achievement, maintenance, avoidance and approach [32]. The inherent nature of this categorisation makes them task-independent. When a robot opens the lid of a bottle, the task is said to be achieved. A mobile robot moving along a marked track is an example of a maintenance task. Reward functions can be based on this categorisation of tasks, thus making them task-independent. In Chapter 4, Section 4.3, this research proposes reward functions based on task types. Using simulated e-puck mobile robot based experiments, Section 4.5 shows how the robot using the proposed reward functions learns to attain maintenance, achievement, approach, and avoidance tasks.

1.3.3 A technique to self-generate tasks of varying levels of complexity

In Chapter 5, Section 5.3, this research proposes a novel task generation technique. The proposal uses agglomerative hierarchical clustering to generate regions within the agent's state space. The number of clusters or regions generated can be varied, generating fewer clusters, i.e. higher-level abstract regions, to more clusters, i.e. granular abstractions. These aggregated state attributes are then used to generate tasks. Furthermore, a change in its environment triggers a continuous learning agent to explore its environment and regenerate the aggregations. These new unique aggregations are integrated within the list of previous unique aggregations, making them suitable for continuous learning. Using simulated e-puck mobile robot experiments, Section 5.4 demonstrates the task generation.

1.3.4 A technique to concurrently compose primitive skills to form solutions for compound tasks

A compound task can be composed of primitive tasks sequenced together or concurrently combined primitive tasks. This research proposes a novel concurrent skill combination technique for the reinforcement learning policies represented by neural networks. Policies combined using this technique provide simplicity and understandability of Q-table based representation and scalability of neural networks. In Chapter 6, Section 6.3, this research proposes the generation of policy for compound tasks using a method similar to the average model weight ensemble [36] [37]. The policy comprises average learnable parameters of the constituent primitive tasks. Using simulated e-puck mobile robot experiments, Section 6.5 demonstrates how the combined primitive policies can be used as a solution for a compound task with little or no additional training.

1.3.5 Metrology for agent performance, task difficulty and agent competency

The metrics generally used to measure a reinforcement learning agent's performance is the reward gained by the agent in each episode. That is appropriate for achievement type tasks, i.e. when the desired state is reached, the episode is considered to be completed. However, maintenance tasks are non-ending, and the concept of the episode is not relevant; thus, the commonly used metric is not an effective way to measure an agent's performance. This thesis proposes new metrics to measure an agent's performance for maintenance, achievement, approach, and avoidance task types. Chapter 4, Section 4.4, details the proposed agent performance metrics. A measure of task difficulty and agent's competency for a skill can be used for task prioritisation and even as intrinsic motivation. Chapter 6, Section 6.4, proposes metrics to measure the task difficulty and agent competency.

1.4 Organisation of the Thesis

This chapter introduces the topic of the thesis, provides the motivation behind this research, details the research questions, and lists the contributions of this thesis. The rest of the thesis is organised as described below and shown in a graphic format in Figure 1.1.

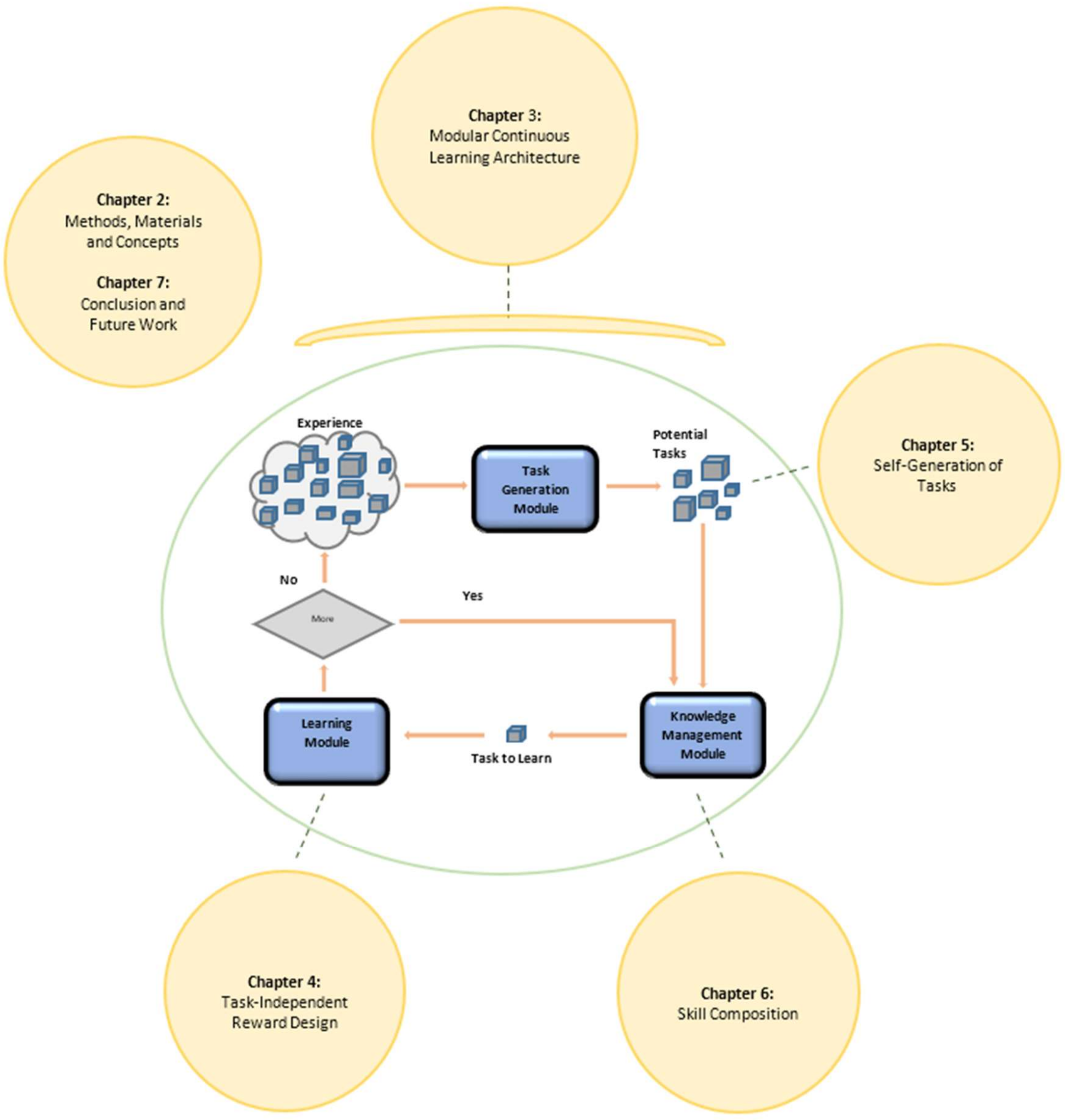


Figure 1.1: A graphical view of the organisation of the thesis.

<u>Chapter 2: Methods, Materials and Concepts</u>

Chapter 2 details the methods, materials and concepts used in this thesis. It starts by describing reinforcement learning concepts and algorithms, followed by the adaptive resonance theory algorithm. It then provides the details of the e-puck mobile robot, followed by other key concepts used throughout this thesis.

<u>Chapter 3: Agent Architecture for Open-Ended and Continuous Learning</u>

Chapter 3 will describe the proposed agent architecture for open-ended and continuous learning, i.e. contribution #1 of this thesis. It will describe the essential components of the architecture and how they are integrated to form a 'Modular Continuous Learning Architecture'. Using simulated e-puck mobile robot based experiments, it will show how the architecture results in agent learning in an open-ended and continuous manner.

<u>Chapter 4: Reward Design for Autonomous Learning</u>

Chapter 4 will describe a novel approach to reward function design, a contribution related to the task learning module. It will detail task-independent reward functions that are based on the type of task. The experiments will use the various types of tasks generated by Merrick et al. [38] and a simulated e-puck mobile robot to demonstrate how it learns to attain those tasks. The learning will be measured using the proposed agent performance metrics. This chapter will also show how the proposed task-independent reward functions can be used to learn compound tasks.

<u>Chapter 5: Self Generation of Tasks to Direct the Learning</u>

Chapter 5 will detail a novel task generation technique capable of generating tasks of varying complexity, a contribution related to the task generation module. It will then detail the results of the experiments related to the proposed task generation technique. Using a simulated e-puck mobile robot, it will be demonstrated how the attributes that form a robot's state space are aggregated to form features that are then used to create tasks ranging from simple to more complex.

<u>Chapter 6: Reuse of Learned Knowledge by Skill Composition</u>

Chapter 6 will detail a novel skill composition technique for the reinforcement learning policies represented by neural networks, a contribution related to the knowledge management module of the architecture proposed in Chapter 3. It will then detail the results of the experiments related to the proposed skill composition technique. Using a simulated e-puck mobile robot, this chapter will demonstrate how the robot can combine primitive skills to form solutions to attain compound tasks with little or no additional training. It will compare those results with the results of learning to attain the compound task from scratch.

Chapter 7: Conclusion and Future Work

Chapter 7 will revisit the research questions, summarise how the contributions of this research answer those questions and extend the current state-of-the-art. The final section of this chapter will list the future direction of this research.

CHAPTER 2 METHODS, MATERIALS AND CONCEPTS

2.1 Introduction

This chapter details the methods, materials and concepts used in this thesis. It starts with reinforcement learning used in the learning module of the architecture proposed in this thesis, followed by the adaptive resonance theory algorithm. Section 2.3 details the simulation software and the mobile robot used in this thesis. Section 2.4 then details other key concepts used throughout this thesis.

2.2 Algorithms

This section will detail the algorithms and fundamental reinforcement learning concepts used in this thesis. The algorithms include reinforcement learning used in all the chapters and adaptive resonance theory used in Chapter 3.

2.2.1 Reinforcement Learning

Reinforcement learning is a method of learning where the agent learns by interacting with its environment. It is not instructed what actions it should take while in a particular situation but must figure out by itself by trying different actions, i.e. the agent learns by trial and error. As shown in Figure 2.1, the agent perceives the state of the environment, takes action in the current state and receives positive or negative feedback called a reward. The agent must compute the most favourable action by selecting and attempting an action from the available set of actions. Over time, reinforcement learning forms a policy, a mapping between states and actions that help decision-making.

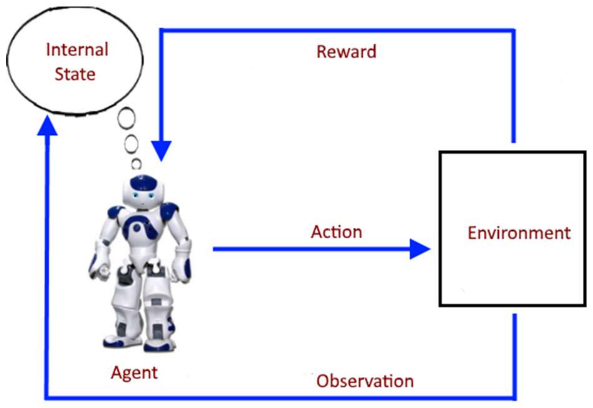


Figure 2.1: Graphic representation of reinforcement learning.

State Space

In reinforcement learning, an agent's state s is a vector of parameters that describes its representation in the environment. That 'state' can be expressed as:

$$s = [\, u^1, u^2, u^3, \ldots u^n] \tag{2.1}$$

where each attribute u^i is typically a numerical value that describes an external or internal variable, and n is the number of attributes of the state. For example, consider a state of a mobile robot represented in terms of its side and front proximity sensors as the state attributes. Further, consider that the state attributes are discretised binary values. Thus, the state can be represented using vector [0 0 0 0]. As will be detailed in the next subsection, this state indicates that all the proximity sensor readings are 0, i.e. no object in close proximity of the mobile robot.

The state space S, also sometimes referred to as an observation space, is a collection of all agent states. The state space can be discrete or continuous. If all the vector attributes that make up the state of the agent are discrete, then the state space is said to be discrete. A discrete state space is represented as shown in the equation below, where v is the number of states in the state space S. If one or more attributes are continuous, then the state space is said to be continuous.

$$S = \{s^1, s^2, s^3, \ldots, s^v\} \tag{2.2}$$

Continuing with the same mobile robot example as above where the state is represented in terms of its side and front proximity sensors and the attributes are discretised into binary values. The state space will be {[0 0 0 0], [1 0 0 0], [0 1 0 0], …, [1 1 1 1]}.

Action Space

The set of available actions is called the action space of the agent. Action space, too, can be discrete or continuous. In the case of discrete action space, represented as shown in the equation below, the agent can select an action from a finite set of m actions.

$$A = \{a^1, a^2, a^3, \dots, a^m\} \tag{2.3}$$

Consider the mobile robot example again. Consider that it can take actions related to turning and moving forward in each of its states. The action space of the robot will be{turn left, move forward, turn right}. Thus, in the state [0 0 0 0] i.e. s^1, it can take either of the actions a^1, a^2, a^3 and similarly in other states $s^2, s^3, \dots, s^v$, it can take those actions.

Reinforcement Learning Representation

The reinforcement learning problem is formulated using Markov Decision Process, whose main components are:

- A set of states $s \in S$, where S is the state space
- A set of actions $a \in A$ the agent can take, where A is the action space.
- A transition function (also referred to as the transition model) $T(s_t, a_t, s_{t+1})$ that defines the transition probability of landing in the state s_{t+1} when an action a_t is taken in the state s_t, that is, $T(s_t, a_t, s_{t+1}) = T(s_{t+1}|s_t, a_t)$
- Feedback received from the environment for taking action a_t in state s_t and landing in the state s_{t+1}, that is, a reward $r(s_t, a_t, s_{t+1})$
- A policy π which determines what action to take in a particular state

Reinforcement learning aims to maximise cumulative reward. Thus, many reinforcement learning algorithms find this policy by estimating a value function that computes how good it is for the agent to be in each state. For example:

$$V_{t+1}(s_t) \leftarrow \max_a \sum_{s_{t+1}} T(s_t, a_t, s_{t+1})[r(s_t, a_t, s_{t+1}) + \gamma V_t(s_{t+1})] \quad (2.4)$$

where V is the value, i.e. the expected utility of state, γ is the discount factor, and t is a time step.

Policy

A reinforcement learning agent is not instructed what actions it should take while in a particular situation but must figure out by trying different actions. In each state, the agent takes action and lands in the next state. Using algorithms such as Q-Learning, the agent determines how good it was to take that particular action. Once the reinforcement learning algorithm converges, i.e. the Q-values stabilise, learning is considered complete. These Q-values help form a mapping between states and actions called policy. This mapping suggests that regardless of how the agent arrived at a particular state, it should take action as per the policy as that action will lead to optimal reward in the future. The policy is akin to skill (detailed in Section 2.4) and is represented as π. The commonly used storage mechanisms for policy are Q-table and neural networks. Both mechanisms allow the storage and recall as required. When the agent has mastered the skill, the policy is said to be optimal policy.

Consider the Q-table shown in Table 2.1. A Q-table is a tabular representation of the state and action pairs where the cells store the Q-values. Q-values, the calculation using Q-Learning detailed in Equation (2.5), denotes the action that should be taken to maximize the cumulative reward, i.e. it encodes the future reward. The following is an example of a Q-table for state space of size v and action space of size 5. The cells show the Q-value that is used to determine how good taking a particular action in each state is. Once the learning is complete, using the Q-values, it should be possible to derive the best action to select in a particular state, i.e. the action with the highest Q-value. Such derived mapping is the policy. For example, consider that the policy uses the epsilon-greedy (detailed in the subsection below) action selection strategy, which selects the action with the highest Q-value in each state. Thus, for the Q-table shown in Table 2.1, the state-action mapping, i.e. the policy would select action 2 when in state 1, action 4 when in state 2, action 1 when in state 3 and continue that for all the states.

Table 2.1: An example of a Q-table. A tabular representation of the state and action pairs where the intersection cell stores the Q-value.

	action 1	action 2	action 3	action 4	action 5
state 1	1.2	4.3	0.4	0.8	3.2
state 2	0.3	6.0	0.1	6.9	2.1
state 3	5.7	2.2	3.2	5.2	1.2
state 4	7.9	3.1	5.7	1.7	8.0
…					
state *v*	0.0	0.0	0.9	6.2	3.4

Action Selection Strategy

A key challenge with reinforcement learning algorithms involves finding the right balance between exploration and exploitation so that it does not converge to a suboptimal solution. Through a trial and error procedure, the agent takes action, records the reward it receives and explores the value function in different regions of the environment. To reach new areas of the state space, reinforcement learning algorithms try out different untried actions randomly. That is called exploration. Too much exploration may prevent maximizing the short term reward because of the lower reward yielded by some actions. On the other hand, the agent can choose to maximize the reward by using the knowledge gained from previous successful actions. That is called exploitation. Too much exploitation, however, prevents the agent from maximizing long term reward because the action chosen may not be optimal. The perceived maxima may be a local maxima.

For Q-table based algorithms, techniques such as the epsilon-greedy and softmax can be used for action selection to balance exploration and exploitation. The epsilon-greedy strategy is used for the experiments in Chapters 3, 4 and 5. The epsilon-greedy strategy has the epsilon parameter that determines the exploration/exploitation ratio, i.e. what percentage of time steps will the agent take random steps to explore new actions versus selecting the best action based on the current policy. The value of the parameter ranges from 0 to 1. For example, when that parameter is set to 0.1 with no decay, 90% of the time, the agent selects the best action according to the current policy; however, 10% of the time, it will randomly select an action. "No decay" indicates that even when the agent has learned, it continues to take random action 10% of the time. The other options are to end the exploration completely when the learning is complete and “linear decay”, where the exploration is gradually reduced as the learning progresses.

Q-Learning

Q-Learning [39] is a reinforcement learning algorithm proposed by C.H. Watkins and P. Dayan. The state-action value function Q estimates the value for selecting an action a_t in a state s_t at time step t is given as:

$$\begin{aligned} &Q_t^{new}(s_t, a_t) \\ &= Q_t(s_t, a_t) + \alpha \left[r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_{t+1}(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)\right] \end{aligned} \tag{2.5}$$

Where $r(s_t, a_t, s_{t+1})$ is the reward received for carrying out an action a_t in the state s_t, α is the learning rate $(0 < \alpha < 1)$ and γ is the discount factor $(0 < \gamma < 1)$. The algorithm is generally used to learn the policy for domains that can be represented by discrete state and action spaces. The algorithm starts by initializing the Q-table and then begins the learning loop. In this loop, the agent takes action in the current state and receives a reward and lands in the next state. This reward is used to update the Q-value as per Equation (2.5), and the process repeats in the next state. The algorithm maintains and updates a Q-table that contains the action values, i.e. Q-value for each state-action pair. It is model-free learning, i.e. it does not use the transition model for learning, and all the learning happens through actual interactions with the environment, which are typically expensive. The core logic of the algorithm, as shown by the equation, is that the Q-function is calculated based on the expected utility for taking the best action in a state and assuming that the optimal policy is followed after that, i.e. the algorithm uses the current estimates of the Q-function to obtain a new (better) estimate of Q.

Dyna-Q

Dyna-Q [40], a combination of Dyna architecture with Q-Learning, was proposed by R. Sutton. In Dyna-Q, the Q-Learning is supplemented with the transition model $T(s_t, a_t, s_{t+1})$ thus combining both model-based and model-free learning, i.e. Dyna-Q has additional internal planning steps to speed up learning. Thus, in addition to the Q-Learning’s loop, the additional steps initialise the transition model, iterate through the internal model (also called hallucinate experience) and update the Q-value based on those imaginary experiences. The Q-table is updated using Equation (2.5), as detailed in the Q-Learning section above. These imaginary experiences are updated using the actual experiences with the environment, creating an internal model that keeps track of the state transitions of the agent and the reward that the agent receives during that transition. Thus the algorithm improves its Q-values using the actual interactions with its environment (expensive) and imaginary experiences generated by the transition probability model (typically not expensive). The number of internal planning cycles is set to be greater than the actual interaction with the environment, thus speeding up learning because, typically, actual interactions can be costly in many cases, such as robotic applications.

Algorithm 2.1: Dyna-Q

```
Initialize $Q(s_t, a_t)$ to 0 for all states $s_t \in S$ and action $a_t \in A$
Initialize the transition model $T(s_t, a_t, :)$ to 0 for all states $s_t \in S$ and action $a_t \in A$

for steps = 1:max_learning_steps

    /* Q learning steps */
    Interact with the environment and perceive the current state $s_t$
    Choose action $a_t$ as per the action selection strategy
    Execute action $a_t$ in the environment
    Perceive the next state $s_{t+1}$ and receive reward $r(s_t, a_t, s_{t+1})$
    $Q_t^{new}(s_t, a_t) = Q_t(s_t, a_t) + \alpha [r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_{t+1}(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$

    /* Dyna-Q specific steps - update the transition model */
    $T(s_t, a_t, 1) = s_{t+1}$
    $T(s_t, a_t, 2) = r(s_t, a_t, s_{t+1})$
    /* Dyna-Q specific steps – internal simulation */
    for simulation_steps = 1:max_simulation_steps
        Randomly select a previously visited state $s_t$ as per the transition model $T$
        Randomly select an action $a_t$ previously taken in that state $s_t$
        Execute action $a_t$ in simulation
        Perceive the next state $s_{t+1}$ and receive reward $r(s_t, a_t, s_{t+1})$ as per the transition model $T$
        $Q_t^{new}(s_t, a_t) = Q_t(s_t, a_t) + \alpha [r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q_{t+1}(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$
    end for

end for
```

A comparison between Q-Learning and Dyna-Q is detailed in Appendix B. Considering the pros and cons of each algorithm and the performance comparison as detailed in Appendix B, Dyna-Q is the algorithm of choice for the experiments in Chapters 3, 4 and 5. Dyna-Q was implemented using MATLAB. The mobile robot used this algorithm to learn skills, i.e. the robot is not instructed using an explicit control program on how to attain the tasks or exhibit certain behaviours but instead learns by itself using reinforcement learning. The number of planning cycles for model learning was set to 25, i.e. before attempting one action with the real environment, the algorithm attempted 25 actions in internal simulation using imaginary experiences.

Advantage Actor-Critic (A2C)

Reinforcement Learning consists of algorithms that are either value-based methods where the agent learns the value function that determines how good it is to take a particular action in a specific state or policy-based methods where the agent directly optimizes the policy by sampling several rollouts of the episode. The actor-critic family of algorithms is a hybrid approach where the critic is trained to estimate the value function and provide feedback to the actor that aims to optimize the policy. When implemented using a neural network, the actor network determines what action to take when in a particular state, and the critic network determines how good that action was and provides feedback to the actor network. The Advantage Actor-Critic (A2C) [41] uses the 'Advantage' instead of the 'Value' function, which leads to learning stability. The advantage is calculated as:

$$Adv(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.6}$$

where $V(s_t)$ is the value at s_t and $Q(s_t, a_t)$ is the Q-value function. In the experiments in Chapter 6, the A2C algorithm from MATLAB's Reinforcement Learning Toolbox was used, and the mobile robot used this algorithm to learn primitive and compound skills.

Option

Reinforcement learning literature also describes the concept of temporal abstractions that take control of the execution for a period, follow a learnt policy and eventually end execution and relinquish control. This abstraction is called an 'option', a closed-loop policy for taking actions over a period [42][43]. Thus, an option is a well-defined macro action denoted by a tuple comprising: an initiation set of states I, termination condition β, and the closed-loop policy π. It is like a subroutine that gets called when the agent is in one of the specific sets of states, i.e. triggered when a particular starting condition is satisfied. When invoked, it follows a learned policy and ends when the termination condition is satisfied and relinquishes control. That allows existing behaviours to be used while learning new behaviours, thus speeding up learning [44].

Intrinsically Motivated Reinforcement Learning

For many dynamic environments, upfront knowledge of the tasks to be learnt is unknown, and hence it is not possible to design the reinforcement learning reward for the task. More recent work has considered the idea of reward modelled using the psychological concept of motivation. Broadly, motivation can be categorised into two types: intrinsic and extrinsic. Intrinsic motivation can be used to model reward functions that can lead to the emergence of task-oriented performance without making strong assumptions about which specific tasks will be learned before the interaction with the environment. When intrinsic motivation is combined with reinforcement learning, it creates a mechanism whereby the system designer is no longer required to program a task-specific reward. Thus, the combination of intrinsic motivation with reinforcement learning results in intrinsically motivated reinforcement learning [45].

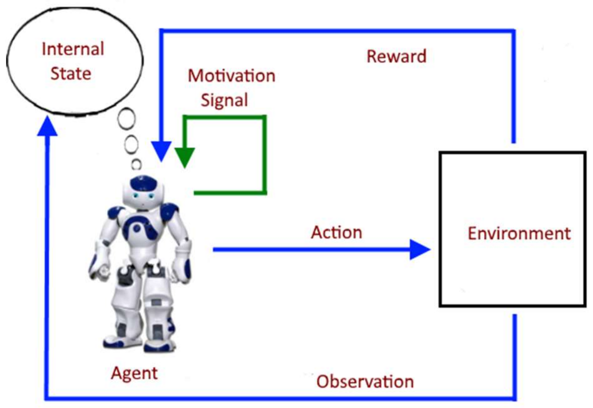


Figure 2.2: Graphic representation of intrinsically motivated reinforcement learning.

Figure 2.2 shows a representation of intrinsically motivated reinforcement learning. Motivation signal is typically computed online as a function of the current experienced state, and some representation of all the states experienced so far, i.e. is independent of task-specific factors in the environment. The signal may drive the acquisition of knowledge or a skill that is not immediately useful but could be useful later on [45]. An agent generates this signal because the task is inherently enjoyable, leading to further exploration of its environment, manipulation/play, or learning the skill.

2.2.2 Adaptive Resonance Theory

Adaptive Resonance Theory (ART) [46] and its variants are neural networks based clustering algorithms. ART1 (used in Chapter 3 for task generation) is an unsupervised learning variant that takes binary vector data points as input. ART1 consists of a comparison field layer, recognition field layer and vigilance parameter. The comparison field layer takes the data points as input. The data points are compared and transferred to the best match in the recognition field layer that represents the category or the cluster. The comparison is based on the concept of 'resonance', which is a similarity measure. If a match is not found, a new category is created. The maximum number of categories is not determined up front but depends on the data points; hence the algorithm is considered 'adaptive'. The vigilance parameter determines how granular the categories are, in turn determining how many categories/clusters are generated. A higher vigilance value generates fine-grained clusters, and a lower vigilance value generates a coarse-grained cluster.

Typically, a task generation process can be divided into the experience gathering phase that collects the data points and the clustering phase that generates potential tasks. During this phase, the states experienced by the agent are recorded. That experience forms the data points for the ART1 algorithm for the clustering phase. ART1 begins by considering the first data point as a new category. It then processes the data points one by one by comparing their similarity with the existing categories. If the data point being processed is a close enough match, it is added to the existing category; else, a new category is created. The granularity and number of categories created depend on ART1's vigilance parameter.

Once the data points have been categorised/clustered, the next step is to select a representative data point. That can be done in several ways (such as selecting a representative or calculating the cluster centroid), with each method having its pros and cons. In the experiments in Chapter 3, the first data point in the category was selected as a representative for simplicity. This representative data point is then considered a ‘potential task’.

2.3 Mobile Robot and Simulation Software

Webots

The simulation software Webots (www.cyberbotics.com) was used to create various environments and simulate the robot’s dynamics. Webots integrates with multiple programming languages enabling the robot control program to be written in any supported language. The experiments in this thesis use MATLAB to implement the reinforcement learning algorithm, which is used as a control program for the simulated e-puck.

E-Puck

An e-puck mobile robot (www.e-puck.org) is used for the experiments throughout this thesis. An e-puck, as shown in Figure 2.3, is a small (~7 cm diameter and ~5 cm height) two-wheeled mobile robot. It is a differential wheeled robot, i.e. the two wheels can be separately controlled. It has eight infrared proximity sensors to measure the robot’s distance from an obstacle. The range of those sensors is 6 cm. It has accelerometers to measure acceleration along all three axes; a 52x39 pixels resolution colour camera, a compass, a microphone and speakers. As an add-on capability, it can be fitted with ground sensors. That provides three infrared ground sensors that can be used to detect edges for fall detection and black/white bands on the ground for applications such as line following.

In Webots, the proximity sensor value for e-puck ranges from 0 to 2000, with a high value indicating that an object is nearby. The ground sensor value ranges from 0 to 1000, with a high reading indicating that the sensor is detecting dark area / black colour on the ground. Using such granular values does not result in any benefit for the experiments in this thesis. In fact, since the Dyna-Q algorithm, which uses Q-table to represent the policy, is used in Chapters 3, 4 and 5, the actual sensor values result in huge state space and corresponding slow learning. Hence the proximity and ground sensor values are discretised. Discretising to 3 (object nearby; not so nearby; far away) or another relatively small number would not result in much detrimental effect from an algorithm performance point of view; however, doing that is not beneficial either. Hence for simplicity, binary discretisation is used in those chapters. Also, for consistency, the binary discretisation is continued in Chapter 6, where the A2C algorithm, which uses a neural network to represent the policy, is used.

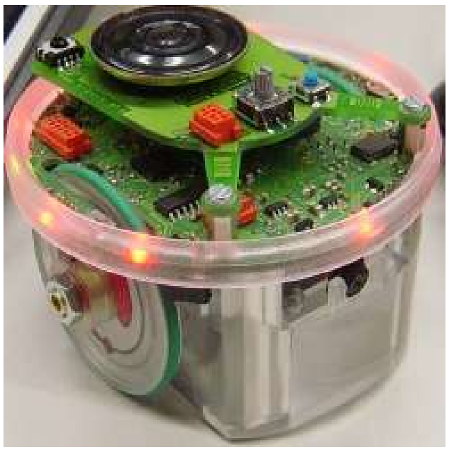


Figure 2.3: e-puck mobile robot

For the proximity sensors, the value 1 indicates that an object is in close proximity, and 0 indicates no object nearby. The proximity sensor's binary value is considered 1 if the actual sensor reading is greater than 500, otherwise considered 0, i.e. if the actual value is greater than 500, the object is considered nearby. This value of 500 was determined based on trial and error to eliminate any false positives due to sensor noise. For ground sensors, the value 1 indicates that the sensor detects black colour, and 0 indicates that it detects white. The ground sensor's binary value is considered 1 if the actual sensor value is greater than 300, otherwise considered 0. Again, this value of 300 was determined based on trial and error to eliminate any false positives due to sensor noise. For simplicity and consistency, binary values are used to represent the left and right wheel speed. A value of 1 indicates that the wheel is moving forward, and 0 indicates that it is moving backwards.

E-Puck Learning a Task using Reinforcement Learning

In the following chapters, a different combination of sensors is used based on the experiments, and each of the chapters details the state representation. Shown here is an example of a state vector used to detail how reinforcement learning agent learns policies for certain tasks. Consider that the left wheel, right wheel, the side and front proximity sensors (labelled in a clockwise direction: *Front-Right, Right-Diagonal, Right, Left, Left-Diagonal, Front-Left*) are used as the attributes to represent the state vector, i.e. the vector can be represented as [ω^L ω^R p^{FR} p^{RD} p^R p^L p^{LD} p^{FL}]. Figure 2.5 is a sketched top view of an e-puck with the labelled wheels, proximity sensors, ground sensors and camera. Consider that the action space consisted of turning left, stepping forward, and turning right. Further, consider that the task is for the e-puck to learn to avoid obstacles in the arena shown in Figure 2.4. That task can be represented as maintaining [1 1 0 0 0 0 0 0] state, i.e. maintaining:

$$max(left\_wheel\_speed) \wedge max(right\_wheel\_speed) \wedge min(all\_proximity\_sensors)$$

The e-puck receives a positive reward in that state and a negative reward in all other states.

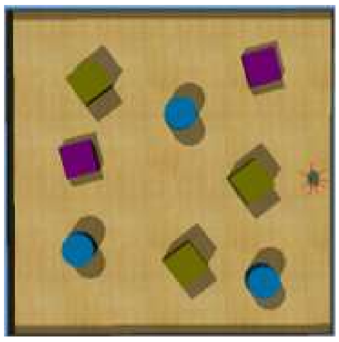


Figure 2.4: Top view of the arena created in Webots.

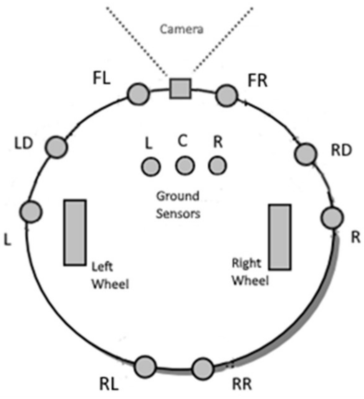


Figure 2.5: A plan view of e-puck with labelled sensors (proximity sensors, ground sensors, and camera) and wheels.

State Vector:

[ω^L ω^R p^{FR} p^{RD} p^R p^L p^{LD} p^{FL}]

Actions:

{
1 - Turn Left,
2 - Step Forward,
3 - Turn Right
}

When e-puck starts learning, it tries different actions in the current state. When an action is taken in a particular state, it lands in a different state. For example, when the e-puck has an obstacle in front, it may try out an action to turn right. That would result in the e-puck having the obstacle to its left, a different state per the state representation. The e-puck may choose to move forward in this state, thus moving away from the obstacle. Over time, by trying such actions, it learns to maintain the desired state, i.e. it learns that when it is near an obstacle or a wall, it should turn away from it to maintain zero sensor values of its proximity sensors and gain positive reward. When the algorithm converges, the skill is said to be learned. When this learned policy is tested, the e-puck will exhibit the behaviour of avoiding obstacles while moving forward.

2.4 Other Concepts used in this Thesis

Task

A task is defined as a piece of work that is attempted. A typical definition of a task is an objective that the agent should achieve [47]. The term 'task' is also interchangeably referred to as 'goal' in the literature. Tasks can range from simple, primitive tasks to complex, composite tasks and have been categorised based on the following criteria: categorisation abstraction from 'high level' tasks covering functional or behavioural aspects to 'low level', concrete tasks that cover the fine-grained definition of what those aspects mean [47] and categorisation from 'hard' tasks, which can be validated straightforwardly, to 'soft' tasks that are difficult to validate [47]. Further, Van Riemsdijk et al. [48] have classified tasks as state-based/declarative, where the task is to reach a specific desired situation, and action-based/procedural, where the task is to execute actions. The declarative tasks are further classified into the 'query', 'achieve' and 'maintain' tasks and the procedural tasks are further classified into 'perform' tasks. That is, tasks have been categorised based on the way they are attained. That categorisation leads to 'maintenance', 'achievement', 'approach', 'test', and 'optimisation' types, to name a few [32].

Skill

A solution to the task is defined as a skill, i.e. learned knowledge. It is the ability of an agent to solve the task and is acquired by training. A primitive skill is a solution to a primitive task, and a compound skill is a composition of primitive skills. In reinforcement learning, a skill is represented as a policy that maps states and actions and is learned by training. Further, the primitive skills can be combined in sequential order or concurrently to form solutions for compound tasks. For a sequential combination, a concept of temporal abstraction called option (detailed above) is commonly used. When multiple such temporal abstractions are sequenced together, it forms a solution for a compound task. Although not a well-researched area, there are mechanisms to combine the skills concurrently. That is further explored in Chapter 6 of this thesis.

Event

When the agent is in a state s_t and takes action a_t, it transitions to a new state. This transition can be called an event E_t and represented as:

$$E_t = [e_t^1, e_t^2, e_t^3, \dots, e_t^n] \tag{2.7}$$

where each attribute that makes up an event is $e_t^i = u_t^i - u_{t-1}^i$ and $s_t = [u_t^1, u_t^2, u_t^3, \dots, u_t^n]$. That is, an event can also be represented as:

$$E_t = s_t - s_{t-1} = [(u_t^1 - u_{t-1}^1), (u_t^2 - u_{t-1}^2), \dots, (u_t^n - u_{t-1}^n)] \tag{2.8}$$

An event, thus, is a vector of difference of the state attributes and models the state transitions caused by the action. In such representation, the event is unaware of any task-specific assumptions about the values of the state attributes, thus making this representation ideal for defining the transition in a task-independent manner [49].

Experience

This thesis defines an experience consisting of three elements: i) the states s_t encountered by the agent, ii) the state transitions or the events E_t and iii) the actions a_t that the agent has performed. The experience, denoted by X, is a trajectory denoted in the equation below. As will be seen later in the thesis, it forms the input data points from which the tasks can be constructed.

$$X = \{s_t, E_t, a_t \mid t = 1,2,3, \dots\} \tag{2.9}$$

There is a similar concept used in reinforcement learning called ‘rollout’. A rollout, however, refers to the trajectory when the agent is attempting to learn a particular task. This thesis uses the term experience to refer to the trajectory of the agent when it is in the body-babbling / experience-gathering phase.

With the algorithms detailed and concepts defined, the next chapter proposes the agent architecture for open-ended and continuous learning.

CHAPTER 3 AGENT ARCHITECTURE FOR OPEN-ENDED AND CONTINUOUS LEARNING

Parts of this chapter have been published in: P. Dhakan, K. E. Merrick, I. Rano, and N. Siddique, "Modular Continuous Learning Framework," 2018 Joint IEEE 8th International Conference on Development and Learning and Epigenetic Robotics, ICDL-EpiRob 2018. Tokyo, Japan, pp. 107–112, 2018.

3.1 Introduction

The focus of lifelong/continuous learning research is to overcome the "Frame Problem" [24], i.e. the focus is on the cognitive aspect of learning and representation of the core commonality of the related tasks to learn skills for multiple tasks. The essential components of such architectures are ways to represent a skill, a memory to store the skills and a mechanism to transfer the skills to solve new tasks [50]. Thus, continuous learning in itself cannot form a complete agent architecture since, in the real world, the environment is dynamically changing, and it is not always possible to determine in advance all the skills that the agent will require [22]. On the other hand, open-ended learning is focused on the meta-cognitive aspects of 'what to learn' [19] and 'when to learn' [20]. That behaviour is envisaged to empower the agent such that it will be able to decide by itself which skills to acquire. Open-ended learning and continuous learning (the two complementary aspects) are required to create a comprehensive agent architecture. The third aspect is learning autonomy, i.e. self-sufficiency, in deciding 'how to learn' the task. Reinforcement learning is a method of learning where the agent learns by interacting with its environment making it a good fit for the required third component of the agent architecture.

The review of the literature on architectures that provide an agent with an open-ended learning capability [12] [51] [52] [53] [54] shows that they have the following modules: (i) a module for task generation and (ii) a module for learning the skills for the tasks generated. Many of those learning architectures also use reinforcement learning as the learning module [12] [51] [52]. However, they lack either the knowledge repository or the continuous learning aspect. The contribution of this chapter is an agent architecture for open-ended and continuous learning. This chapter will show how open-ended learning, knowledge store and continuous learning are combined to form the modular learning architecture. Using a simulated e-puck mobile robot, the experiments in this chapter will demonstrate the validity of the architecture. It will show that the e-puck when placed in different arenas, autonomously generates new tasks and then learns skills to solve them—thus demonstrating the capability of autonomously learning new skills in an open-ended way.

The rest of this chapter is organised as follows: Section 3.2 reviews the literature on agent architectures for autonomous, open-ended and continuous learning. Section 3.3 details the proposed agent architecture. Section 3.4 elaborates on the experimental setup and results of the experiments demonstrating open-ended and continuous learning. Finally, Section 3.5 provides concluding remarks.

3.2 Agent Architectures for Open-Ended Learning

Broadly, agents are classified to be reflex, goal-oriented, adaptive and autonomous [6]. An autonomous agent, a broad classification in itself [55], can sense its environment and take action to accomplish a specific goal. That category is further classified into several subcategories, of which executive autonomy is one that is related to the setting and execution of goals [28]. In the last fifteen years, motivated reinforcement learning is emerging as a popular way to achieve such autonomy [56]. Motivated behaviour is argued to be crucial for an agent to gain the competence that is essential for autonomy [45]. This competence is gained in a task-independent manner [22], resulting in the development of an entity capable of accomplishing varied activities compared to an agent capable of accomplishing only one specific or a few related activities [57] [58]. The following subsection will review motivated reinforcement learning agent architectures.

3.2.1 Motivated reinforcement learning agent architectures

Reinforcement learning is a form of learning in which an agent learns by interacting with its environment [40]. The agent is not provided with a rule book, instructions on what to do in a particular situation or prior knowledge of its environment but must figure it out by itself by trying out the available actions. The agent senses its environment, takes action and receives feedback from the environment, called ‘reward’. This reward guides the learning by providing the agent with a sense of which actions to take in different situations. The broad aim of the reinforcement learning agent is to maximise this cumulative reward. In early work, the reward was typically defined to be specific to a particular task. However, for many dynamic environments, it is not known upfront which tasks are to be learned. As a solution, the idea of reward modelled using the psychological concept of motivation has gained popularity, and in the last decade, intrinsically motivated reinforcement learning has attracted particular interest. When motivation is combined with reinforcement learning, it creates a mechanism whereby a task-specific reward is no longer required to be programmed. It creates a motivated reinforcement learning agent that can select a task to be learned [11] and learn that task autonomously. That results in an autonomous learning agent that can learn complex behaviours in a task-independent and open-ended way.

Figure 2.1 shows a general intrinsically motivated reinforcement learning architecture in which, in addition to the environment provided feedback, the agent internally generates a signal, and in combination, both of those form the basis for its actions. The intrinsic motivation signal can either be combined with the extrinsic reward using certain criteria or can be used exclusively, that is, instead of the extrinsic reward. Broadly speaking, the architecture introduces a meta-learning layer, where the role of the motivation signal is to enable the learning algorithm to focus the learning [45]. These software architectures are essentially formed by combining the basic building blocks of reinforcement learning algorithms and the blocks that generate motivation signals. The different intrinsically motivated reinforcement learning architectures detailed in this section are then formed by combining the motivation generator block with different types of reinforcement learning algorithms and blocks responsible for managing the learned skills. The subsections below describe the flat, hierarchical and multi-option architectures found in the motivated reinforcement learning literature.

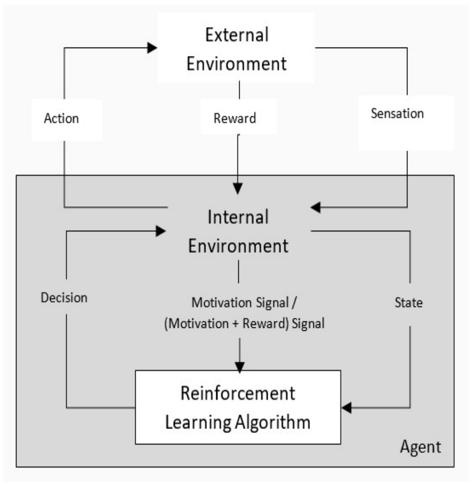

External Environment
Action
Reward
Sensation
Internal Environment
Decision
Motivation Signal / (Motivation + Reward) Signal
State
Reinforcement Learning Algorithm
Agent

Figure 3.1: Concept of intrinsically motivated reinforcement learning. The reward received by the reinforcement learning agent is a combination of external reward and motivation signal.

Motivated Flat Reinforcement Learning: Perhaps the most basic variant of intrinsically motivated reinforcement learning [45] is where only one policy is learned. This variant is termed motivated flat reinforcement learning (MFRL) by Merrick and Maher [44]. MFRL combines motivation with a single non-hierarchical (or 'flat') reinforcement learning algorithm forming an adaptive architecture for performing multiple tasks. Because MFRL only learns a single policy, that policy adapts to represent a different behaviour at different times [44]. That leads to an advantage in a highly dynamic environment. MFRL is arguably the simplest form of motivated reinforcement learning and is well represented in Figure 2.1. The downside of MFRL is that it does not implement the option; hence there can be no recall of skills. Each learned skill is overwritten when a new skill is learned. Alternatives to MFRL discussed in the following sections overcome this weakness.

Motivated Multi-Option Reinforcement Learning: When an option is combined with reinforcement learning, it creates multi-option reinforcement learning. Knowledge is accumulated by continuously creating additional options representing a solution to a task. When motivation is combined with multi-option reinforcement learning, it creates motivated multi-option reinforcement learning (MMORL). In the case of MMORL, the reinforcement learning block is the multi-options reinforcement learning algorithm. MMORL learns multiple skills, each implemented using options. For each skill, an initiation state triggers the respective option. The policy of that option guides it to reach the termination state, which accomplishes a task. Learning how to accomplish each of the *N* possible tasks is going to be time-consuming, but this will still be cheaper than *N* times the work of learning to achieve a single task [59]. MMORL can be equated to the implementation of multiple motivated flat reinforcement learning with a parent layer triggering the recall of the relevant option. The additional meta-layer is responsible for the addition and deletion of the options. Thus the agent implemented with multi-option reinforcement learning is able to achieve multiple goals. This architecture provides a recall of multiple options at the same level, whereas motivated hierarchical reinforcement learning, detailed in the following subsection, provides a recall of options at a different level, all arranged in a hierarchy.

Motivated Hierarchical Reinforcement Learning: A hierarchical reinforcement learning algorithm is one in which a policy can be decomposed into a hierarchy of sub-policies. This enables the reuse of policies to form a solution to a more complex problem. Similar to multi-option reinforcement learning, hierarchical reinforcement learning is also implemented using options. An option can invoke other options as actions, thus leading to a hierarchical structure of learning and recalling the learned behaviours. When motivation is combined with hierarchical reinforcement learning, it creates motivated hierarchical reinforcement learning (MHRL) [44]. MHRL, like motivated multi-option reinforcement learning, can learn multiple skills. However, the skills are hierarchical skills, arranged from the most basic skills at the bottom-most level of the hierarchy to the complex skills that build upon these basic skills at the higher level, ordered by increasing complexity. Likewise, the skill recall in the case of MHRL is hierarchical too. Thus the MHRL architecture can resolve fairly complex tasks.

Appendices Section A.1 compares the different motivated reinforcement learning architectures in detail. For completeness, this thesis also compares performance measures of these architectures found in the literature, which are detailed in Appendices Section A.2. However, none of the architecture detailed fulfils the criteria for open-ended continuous learning architecture. That is because the focus of these architectures is on learning autonomy, i.e. automation of ‘how to learn’, and they fail to consider the open-ended learning aspect of ‘what to learn’, i.e. what all tasks the agent should be learning. The next subsection reviews the architectures that overcome that limitation.

3.2.2 Goal-oriented autonomous agent architectures

While the focus of the architectures in the previous subsection was on generating task-independent reward functions, the architectures detailed in this subsection focus on task generation to direct the learning.

Goal Discovering Robotic Architecture for Intrinsically Motivated Learning (GRAIL): Santucci et al. proposed GRAIL architecture [12] [60] and, subsequently, an extension of C-GRAIL that is context-aware for the cases where the solution to the same task may require different skills [61]. In this multi-layered architecture, the agent, a simulated iCub robot, detects changes to its environment (considered tasks) and forms their representations for storage. Another layer of architecture then generates a competence-based signal based on the learning progress to select a task from a list of tasks. The agent then aims to learn a solution to achieve that task. Santucci et al. demonstrate how the architecture generates tasks and then autonomously determines which iCub's arms to use to reach a specified object. The architecture, however, lacks an organised mechanism to store and recall the learned knowledge, one of the essential requirements for lifelong learning architecture.

Motivated Introspective Reinforcement Learning: Merrick [11] introduces the concept of combining introspection with intrinsically motivated reinforcement learning. The architecture introduces a meta-layer of introspection that guides the motivated reinforcement learning framework using an options model to activate a skill, suspend the activation, and delete a skill. This motivated introspective reinforcement learning (MIRL) architecture incorporates a task life cycle model with motivated reinforcement learning, thus removing the common reinforcement learning assumption of a fixed set of tasks. With such an architecture, an agent can create tasks online and automatically decide which skills to learn and when. Introspection is used to create and delete options in relation to the discovery of skill acquisition. Here too, the option is used to learn skills that achieve those tasks. Similar to the motivated multi-option reinforcement learning detailed in the previous subsection, it uses a multi-option reinforcement learning algorithm. However, unlike motivated multi-option reinforcement learning, it has a meta-layer, i.e. an introspective layer that manages the life cycle of the skills and provides a mechanism to add, update and remove the skills. The skill recall scalability of MIRL will be comparable to motivated multi-option reinforcement learning [11]. Figure 2.2 shows the MIRL architecture. The additional introspective learning and goal management layers are responsible for managing the life cycle of the skills, i.e. autonomously selecting when to learn a skill to achieve a goal, when to activate and deactivate a particular skill and when to delete a skill. The reinforcement learning agent is implemented using a multi-option reinforcement learning algorithm. The rest of the components in this architecture are the same as the motivated flat reinforcement learning, i.e. the agent senses the environment (represented as a state), takes action, and receives a reward and a motivation signal.

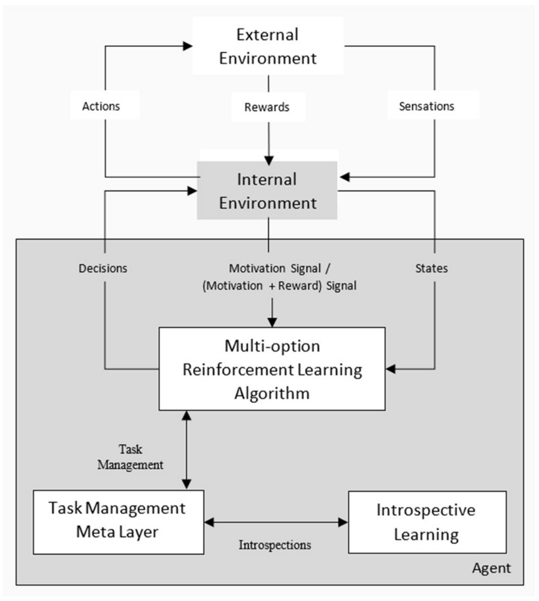


Figure 3.2: Motivated introspective learning; diagram adapted from Merrick [11]

Self-Motivated Incremental Learning (SMILe): Bonarini [52] proposed SMILe, an intrinsically motivated learning architecture that generates a hierarchy of tasks that are then learned using a three-phase process. In the first phase, termed a babbling phase, the agent explores its environment to create a state transition model. In the second phase, the architecture identifies the tasks, i.e. interesting events, using intrinsic motivation. In the final phase, the agent then learns a solution to a task. This architecture lacks a mechanism to store and recall the learned knowledge, thus not satisfying all the criteria for an open-ended continuous learning agent architecture.

Jaidee et al. [62] proposed a task-driven architecture with a focus on task formulation and task management. The agent learns using case-based reasoning and reinforcement learning. The task formulation process starts with the agent comparing observation to expectation. If a discrepancy is found, it tries to reason based on the existing knowledge of case-based reasoning. If the discrepancy is not resolved, it is considered a new task for which the skill should be acquired. The new tasks are added to the set of tasks pending to be learned. The task manager selects a task based on priority and assigns it to the learning algorithm.

Hanheide et al. [63] present an architecture that autonomously generates tasks to demonstrate how task-directed behaviour is more adaptive in dynamic environments. In their experiments, a mobile robot explores an unknown environment, generates a map and then categorises the rooms, which is the task. The architecture's main feature is a knowledge management module that activates and suspends the tasks and prioritises them based on their importance. This architecture can autonomously generate tasks in a new environment and manage those tasks; however, it lacks a learning module to acquire skills to solve those tasks, thus not satisfying all the criteria for an open-ended continuous learning agent architecture.

3.2.3 Other architectures

In addition to the architecture categories mentioned above, several more architecture categories exist. The most popular is cognitive architecture. Broadly the main focus of such architectures is on problem-solving and knowledge management and lacks the mechanism to self-generate tasks. This subsection reviews some of the architectures from this category. However, the scope of the review in this thesis is limited to architecture categories that are an extension of motivated reinforcement learning and focus on either learning autonomy, open-ended learning or continuous learning aspects.

SOAR Cognitive Architecture: Laird [64] proposed architecture for agents dealing with an uncertain and dynamic environment. The architecture has a procedural memory consisting of if-then-else rules as well as a reinforcement learning based module. The permanent memory stores the world model, primitive skills and broad general knowledge. Laird states that architecture enables building knowledge over time and can be pre-seeded and shared by multiple robots. The architecture stores snapshot of experiences that guides future behaviour. The learning can be with or without reinforcement learning and can be using the stored snapshot of experiences or direct interaction with the environment. This architecture has the memory to store knowledge and the capability to learn new skills; however, it lacks the continuous stream of tasks that the agent can aim to learn, thus falling short of fulfilling all the criteria of open-ended continuous learning architecture.

Deferred Restructuring of Experience in Autonomous Machines (DREAM): Doncieux et al. [65] proposed a modular cognitive architecture with open-ended learning capability. Tasks are generated using the babbling approach, where the agent explores its environment to form a list of objects that it can grasp (one task) and objects that it can push (another task). Multiple tasks can be learned, with the learning in the wakeful state, where the agent interacts with the environment and dreaming state, where the agent mulls over the learning without any interaction with the environment, making the architecture suitable for robotic applications since learning can be time-consuming if the robot has to interact with the environment constantly. The architecture also has a cognitive module, a knowledge store that enables the transfer of skills between tasks. From a basic building blocks point of view, this architecture is similar to the architecture proposed in this thesis.

3.2.4 Gap in the state-of-the-art

It is evident from the literature review that progressing reinforcement learning beyond single-task learning and that too, in an open-ended manner, is an active research area. As summarised in Table 3.1, motivated reinforcement learning architecture focuses on learning autonomy such that learning the tasks does not require handcrafted reward functions. On the other hand, goal-oriented agent architectures focus on task generation so that agents can self-direct their learning. Thus, neither of the architectures by themselves form an agent architecture capable of open-ended, continuous and autonomous learning. That raises the question, what are the modules of an open-ended and continuous reinforcement learning architecture?

Table 3.1: Table showing the focus areas of each architecture category reviewed in this literature review section.

Focus area	**Motivated reinforcement learning agent architectures**	**Goal-oriented autonomous agent architectures**
Learning autonomy	✓	✗
Task generation (i.e. 'what to learn')	✗	✓
Continuous learning	✗	✗

While neither of the architectures mentioned above satisfies the criteria, they can be combined since they are complementary. It creates an architecture that enables the agent to self-direct its learning and also enables it to learn the skills with little to no external intervention. The following section explores this further and proposes an architecture with a mechanism to self-discover tasks to learn, a repository to manage the tasks, and a mechanism to learn self-discovered tasks.

3.3 Modular Continuous Learning Architecture

Tasks direct the learning as they are the reason for the agent's actions [32]. For an autonomous agent, a well-formed curriculum is vital for it to learn new and complex skills. Thus, task generation is an essential component of agent architecture, with the primary responsibility of generating tasks for the agent. However, the task generation mechanism may not provide any information regarding the usefulness or similarity of the task with the other tasks, thus leading to a proliferation of tasks that could hinder as opposed to direct the agent's learning. Thus, task management is an important consideration. The task management component can be responsible for keeping track of the tasks that are yet to be learned, similar tasks, and tasks that are not useful or obsolete tasks and hence need to be pruned. It can also be responsible for prioritising the tasks based on the current skill level of the agent. That is especially useful in the case of robots with limited computational resources, as learning every skill from scratch can be very time-consuming. The third essential component is the learning algorithm, enabling the agent to acquire the skill for the prioritized tasks. Thus, to create a comprehensive continuous learning architecture capable of open-ended learning, task generation, task management, and learning algorithm are the main components.

This chapter proposes an agent architecture that satisfies all those criteria. It proposes a generic, domain-independent, and modular architecture termed 'Modular Continuous Learning Architecture' capable of open-ended learning. The following are the essential modules:

1. 'Task Generation Module', that generates the tasks in an open-ended manner,
2. 'Knowledge Management Module', which is a repository of skills, and
3. 'Learning Module', that learns skills to solve those tasks.

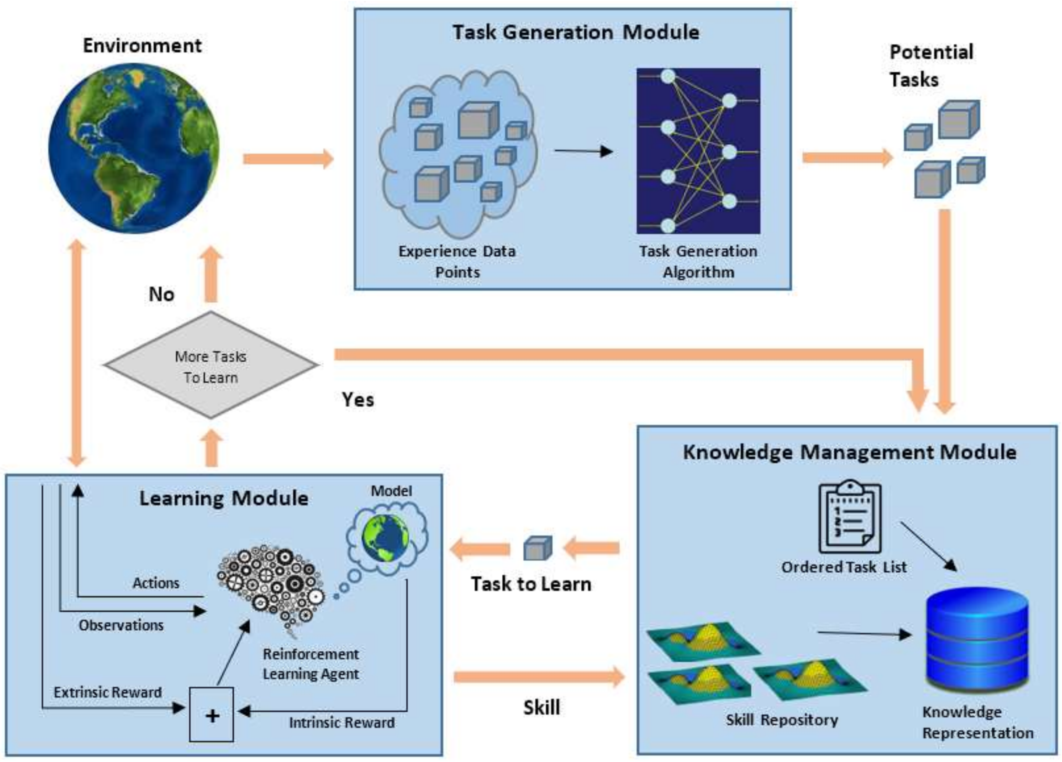


Figure 3.3: Modular Continuous Learning Architecture.

This task generation and skill acquisition are implemented in a continuous loop. That enables the agent to adapt to the dynamic changes in its environment and continuously improve its skills in an open-ended manner. Figure 3.3 shows the proposed Modular

Continuous Learning Architecture that forms the basis for this thesis's work. The following subsections discuss each of the essential components of this architecture in detail.

3.3.1 Task generation module

The 'Task Generation Module' is responsible for directing the agent's learning by autonomously generating a list of tasks to learn. This module makes the architecture capable of open-ended learning and enables the agent to self-direct its learning. It enables the agent to learn more than one task, makes it more adaptive to dynamic environments and is required for it to behave autonomously in the real world. The task generation process, in most cases, begins with the agent gathering the experience by moving about randomly in its environment. That is the same as body babbling described in the developmental robotics literature, where the agent aims to discover its body and its relationship with the environment. That experience is then used to generate a set of potential tasks. The literature review shows that task generation is an active research area and that there are several ways to generate tasks. Tasks can be generated in one or more of the following ways:

- Using artificial curiosity, seek novel situations that form the potential tasks.
- Using the agent's previous experience as data points, generate clusters of unique situations using an unsupervised learning algorithm. The cluster centroids form potential tasks.
- In its simplest form, an externally supplied domain-dependent list of tasks.

The types of tasks generated depend on the implementation of this module, i.e. the module can be designed to generate: i) flat tasks, i.e. non-hierarchical tasks, ii) hierarchy of tasks ranging from top-level tasks to its corresponding sub-tasks, iii) a curriculum of related tasks ranging in complexity from primitive to compound tasks. Typically, the curiosity-based implementations would generate single level tasks. To generate a hierarchy of tasks, options discovery methods [66][67] can be used, and methods such as the one detailed in Chapter 5 can be used to generate a curriculum of tasks.

3.3.2 Knowledge management module

The ‘Knowledge Management Module’ is a knowledge repository. Its primary role is to manage the tasks and store the learned skills. For certain environments, a vast number of tasks can be generated by the task generation module, and it is not always possible or practical to learn all the tasks, especially for robotic applications. Hence, there should be a mechanism to ascertain the task’s usefulness and gauge the task’s similarity with other tasks, both of which may require domain knowledge. Based on that, it should prune and prioritize the tasks. The module should be able to store the skill and recall it when required. Also, it should be able to combine the skills to form solutions for compound tasks. Considering that, the primary responsibilities of this module are:

- Add and delete the tasks to its internal list.
- Maintain the status of the task indicating whether it is learned or not.
- Update the priority of tasks based on similarity with the learned tasks.
- Store and recall the skills.
- Generalize the skills to form a consolidated knowledge representation.

In addition to the above, in most cases, the ability to unlearn or forget a skill is an important consideration for real-world applications. That will be required if the task is no longer valid or an alternative way of attaining the task is learned—the deleting of skill results in freeing up memory and optimising the skill search. Also, task prioritisation is vital as learning every skill is not possible, especially for robots. Such prioritisation can be implemented based on a similarity index generated by the task clustering algorithm [68], novelty-based motivation [11] or based on the current competency level of the agent for similar skills. For example, consider that the task generation module is implemented using a clustering algorithm. Such algorithms internally use similarity indices to generate clusters, i.e. tasks in this case. Those indices can be used to determine the similarity of a newly generated task with other tasks in the knowledge management module. The implementation of the knowledge repository can be based on reinforcement learning packaged policy per skill rather than the approach of one large neural network. That also enables easier storage, recall and composition of the skills as required in the future.

3.3.3 Learning module

The ‘Learning Module’ is responsible for learning the skill required to solve the task. This thesis started with the premise that reinforcement learning is suitable for autonomous learning; hence this module should be implemented using one of the reinforcement learning algorithms. Section 3.4 shows how this module can be implemented using the Dyna-Q algorithm. In reinforcement learning, the agent learns interactively. It perceives the state of its environment, takes action from all the available actions that it can take in that state and receives feedback called ‘reward’. Over time reinforcement learning forms a policy. The learning in reinforcement learning is guided by reward, either received when a milestone is reached or for every step. Reinforcement learning aims to maximize this cumulative reward. Once a task is learned, or the learning cycle is finished, the learning module receives the next task to learn from the ‘Knowledge Management Module’ that maintains a prioritized list of tasks. The outcome of the learning process can be that the task is learned or that it is not learned in the given time frame. If the task is learned, the policy, i.e. the skill, is sent to the ‘Knowledge Management Module’ for storage. The reason a task cannot be learned can be either that the task is too difficult to learn at this time or because it is invalid. The knowledge management module records this.

3.3.4 Continuous learning

The cycle of task generation, skill learning, and knowledge management is shown in Figure 3.3 and detailed in Algorithm 3.1. The knowledge management module, which maintains the list of tasks, supplies the learning module with a task to learn. When all the tasks from that list are learned or when a change is detected in the environment, the task generation module generates a new set of tasks. Thus, there are two loops. One is an internal loop between the knowledge management and learning modules. That loop ensures that all the tasks that are not yet learned are learned. The other loop is an outer loop that checks if there is any change to the environment, and if so, the task generation module generates new potential tasks. That loop can be beneficial if the agent’s environment is dynamically changing. These cycles continue for the whole lifetime of the agent, forming continuous learning. Also, task generation and learning can happen concurrently, with one agent scanning the environment and generating new tasks while the other agent is learning the skills for those tasks. This continuous learning results in an ever-increasing knowledge base of the system.

Algorithm 3.1: Modular Continuous Learning

```
Repeat

  /* 1a - Experience gathering phase */
  for steps = 1: max_exploration_steps
    Interact with the environment using exploration policy
    Gather experience
  end for

  /* 1b – Task generation phase */
  Cluster the tasks using the experience as the data points

  /* 2 – Knowledge management phase */
  From the list of potential tasks, add unique tasks to the task_list
  Sort the task_list

  /* 3 - Learning phase */
  for task = 1:task_list
    for steps = 1:max_learning_steps
      Interact with the environment according to epsilon-greedy
      Simulate and update Reinforcement Learning Q-table using Dyna-Q
    end for
    if Q Learning has converged
      Mark the task as learned
      Store the learned policy for the task
    end if
  end for

until forever
```

3.3.5 Architecture extensions

The flexibility of the architecture allows multiple agents to collaborate in order to generate tasks. Similarly, multiple agents can collaborate to learn those tasks. Figure 3.4 shows the architecture with the task generation and the learning modules with multiple blocks stacked on top of each other, indicating that they can be implemented as multi-agent modules. All the skills are stored in a single repository, thus enabling the transfer and sharing of knowledge between agents. Such transfer between the agents can be advantageous when using simulated and real robots in tandem with the simulated robot carrying out activities that can be too risky or time consuming for the real robot.

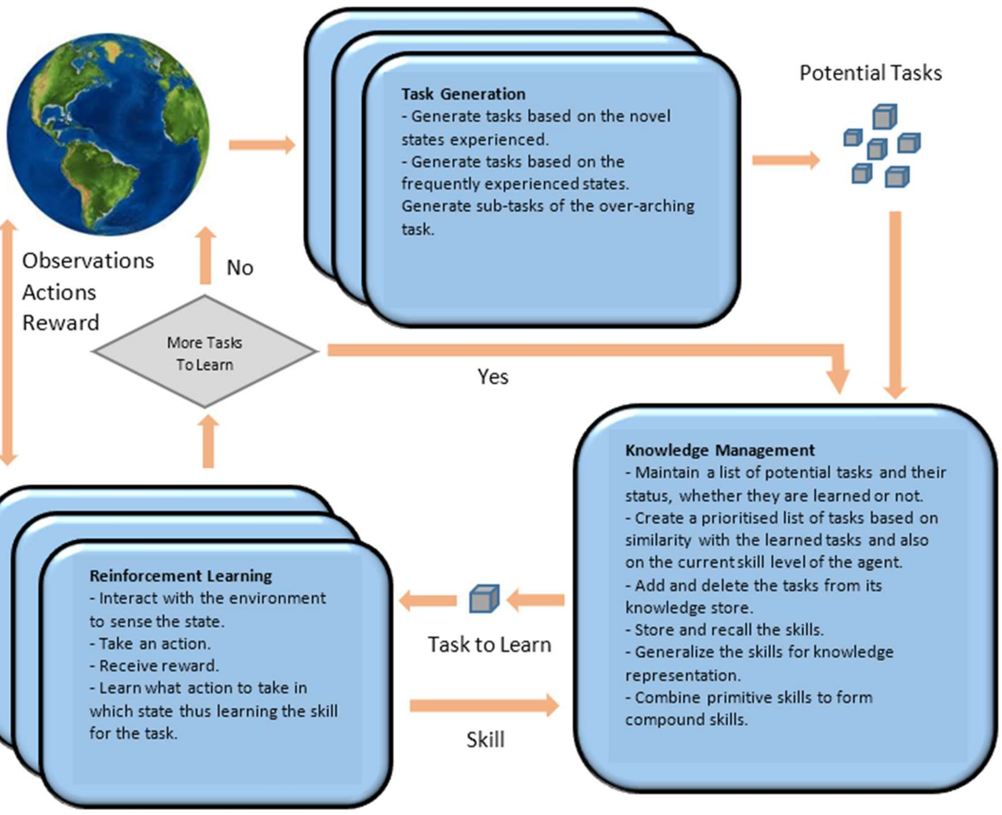

Task Generation
- Generate tasks based on the novel states experienced.
- Generate tasks based on the frequently experienced states.
Generate sub-tasks of the over-arching task.
Potential Tasks
Observations
Actions
Reward
No
More Tasks To Learn
Yes
Knowledge Management
- Maintain a list of potential tasks and their status, whether they are learned or not.
- Create a prioritised list of tasks based on similarity with the learned tasks and also on the current skill level of the agent.
- Add and delete the tasks from its knowledge store.
- Store and recall the skills.
- Generalize the skills for knowledge representation.
- Combine primitive skills to form compound skills.
Reinforcement Learning
- Interact with the environment to sense the state.
- Take an action.
- Receive reward.
- Learn what action to take in which state thus learning the skill for the task.
Task to Learn
Skill

Figure 3.4: A consolidated view of the Modular Continuous Learning Architecture showing functional details of each module and multi-agent capability of the Task Generation Module and the Learning Module.

The literature review detailed the following two categories of open-ended learning agent architectures. The first category was motivated reinforcement learning agent architecture with architectures such as motivated multi-option reinforcement learning (MMORL), where motivation is used to design a task-independent reward. In the architecture proposed in this chapter, the learning module can be implemented with a motivation based task-independent reward design. The second category of agent architecture was goal-oriented autonomous agent architectures with architectures such as goal discovering robotic architecture for intrinsically motivated learning (GRAIL), where motivation is used to generate tasks that direct what the agent learns. In the architecture proposed in this chapter, the task generation module can be implemented using novelty detection, for example, as shown by Marsland et al. in [69], where a neural network can be trained to detect novel perceptions, and such perceptions can then be considered as tasks. In addition, the knowledge management module can be implemented using a competence-based task selector where the system selects which task to learn (from the list of tasks provided by the task generation module) based on the difficulty level of the task and the current competency level of the agent for a similar task. When motivation is used in the design of either of the modules of the proposed architecture, it creates an architecture that can be termed as "motivated open-ended continuous reinforcement learning agent architecture".

Figure 3.4 shows a consolidated view of the architecture with each module's multi-agent capability and functional details. Each module of the architecture also represents a phase of the whole learning cycle. These phases of task generation, learning and knowledge storage can be explicit stages or implemented to work in a continuous way where the agent receives a continuous stream of experience akin to learning online. The architecture allows flexibility in terms of the addition of other external modules, each of which could be responsible for a particular functionality. Also, since the architecture is designed to have low coupling between the modules, there is flexibility in terms of the techniques used to implement each of the modules.

3.3.6 Architecture applications

The proposed agent architecture is envisaged to be applied to robotics, in particular developmental robotics. Different robots have different sensors and actuators, providing them with different capabilities. They locomote and interact with the environment differently. For these robots to be useful, the typical approach is for a designer to write a control program for these robots to be able to carry out specified tasks. However, for dynamically changing environments, even the designer will not know upfront what tasks the robot will need to carry out. Thus, it is difficult to determine upfront what skills the robot will require, and hence not possible to write the control program corresponding to those skills. Moreover, these control programs are specific to the capabilities of the robot's sensors and actuators. Any change to the hardware requires rewriting/optimising the control programs. Developmental robotics' approach to this is to endow the robot with the capability of open-ended and lifelong learning of new skills of increasing difficulty. The architecture proposed in this chapter is a generic architecture with the required modules to empower the robot to accomplish such open-ended and lifelong learning. The architecture, however, does not have modules that allow the robot to learn language-based communication skills; thus, the scope of this architecture is limited to learning sensorimotor skills.

In developmental robotics, the robot is not provided with any innate knowledge. It starts with body babbling to acquire knowledge of its body and environment, creates sensorimotor associations and determines what to learn. The architecture's task generation module is proposed with just that in mind. Further, in developmental robotics, the robot should be able to learn the skills by itself, store and recall skills as required and progressively increase its overall knowledge. The architecture's knowledge management and learning modules are proposed with just that in mind. To put it in perspective, the tasks generated by the robot implemented using the proposed architecture will be dependent on their sensors' and actuators' capabilities (one such implementation is detailed in Chapter 5). The architecture enables the robot to learn skills to accomplish those tasks autonomously. The learned skills can be stored and recalled when required. They can be combined to generate solutions for more complex tasks (one such implementation is detailed in Chapter 6).

The learned behaviour exhibited by the robot will be as per its inherent capabilities depending on the types of sensors and actuators that make up the robot, thus demonstrating skills acquisition in an open-ended manner. During such execution mode, the architecture can be considered as a reactive robot architecture where the learned primitive skills are equivalent to the layer/module of a reactive architecture, such as the subsumption architecture [70]. In such reactive architectures, the world is considered the best model, i.e. there is no internal representation of the external world, or that representation is just a current estimate of the world. The behaviour exhibited by the robot is based on a mapping from state to action, i.e. the learned reinforcement learning policy for a task in the case of the proposed architecture. Such a policy can be triggered based on specific criteria exhibiting the learned behaviour. Also, the primitive skills can be combined (one such implementation is detailed in Chapter 6) to execute a more complex behaviour.

Take, for example, a vacuum cleaning robot with primitive tasks such as (i) detecting the dirt, (ii) cleaning the dirt, (iii) avoiding obstacles, and (iv) detecting an edge of the floor to keep the robot from falling off the stairs. The task generation module will generate such tasks, and the skills will be learned and the learned policy stored in the knowledge management module. During the execution mode, the primitive task of avoiding an obstacle is triggered when the vacuum cleaning robot detects an obstacle in its proximity. Similarly, the primitive task of cleaning the dirt is triggered when the robot detects dirt on the floor. For cases where multiple tasks are triggered simultaneously, such as (a) the robot detecting the dirt while avoiding obstacles and (b) the robot cleaning the dirt while avoiding obstacles and avoiding falling off the stairs, the skill is generated by combining the policies of the constituent primitive tasks.

The following section details the experiments that validate the claims of the architecture. Those experiments use a mobile robot; however, the concept can be extended to robots of other types. Some of the challenges related to such an extension are discussed in the future work section in Chapter 7.

3.4 Mobile Robot Experiments

The previous section proposed an open-ended continuous learning architecture. The experiments in this section validate the basic claims of the architecture—i.e. the open-ended and continuous learning aspects. The scope of the experiments is limited to validating the end-to-end working of the Modular Continuous Learning Architecture.

3.4.1 Objectives of the experiments

The objectives of the experiments in this section are:

- Using a basic task generation mechanism, verify identified tasks that the agent architecture can learn in an open-ended manner.
- Verify, by observing learning progress, that the agent architecture enables continuous learning.

3.4.2 Methods and materials

The experiments in this chapter use a simulated e-puck mobile robot. The simulation software Webots was used to create various environments and simulate the robot's dynamics. This section describes an implementation of each component of the Modular Continuous Learning Architecture described in the previous section.

Robot and its Environment

In the experiments in this chapter, only proximity sensors and ground sensors of the e-puck mobile robot were used. As seen in Figure 3.5, the eight proximity sensors are labelled in a clockwise direction as *Front-Right, Right-Diagonal, Right, Rear-Right, Rear-Left, Left, Left-Diagonal, Front-Left*. The three ground sensors are labelled as *Left, Centre, Right*. The red directional lines in Figure 3.5 show the direction in which the proximity sensors measure the distance to an obstacle. An abbreviated name of each sensor is shown beside the directional lines.

In the experiments in this chapter, binary discretisation was used for state space attribute values, as detailed in Chapter 2. The action space constituted of the following three actions: (i) Turn Left, (ii) Step Forward, and (iii) Turn Right. There was no action representing standing still or moving backwards. The proximity and ground sensors and the left and the right wheels form the state vector of the e-puck, represented by $[\omega^L\ \ \omega^R\ \ p^{FR}\ \ p^{RD}\ \ p^R\ \ p^{RR}\ \ p^{RL}\ \ p^L\ \ p^{LD}\ \ p^{FL}\ \ g^L\ \ g^C\ \ g^R]$. ω^L and ω^R are the motion direction of the left and the right wheels, respectively.

For the experiments, three environments were created in Webots. The arenas, shown in Figure 3.6, Figure 3.7 and Figure 3.8, were 2m x 2m in size. Figure 3.6 shows a top view of the arena with obstacles with a few cylindrical and cuboid objects randomly scattered in the arena. This environment can provide an opportunity to exhibit behaviour, such as avoiding obstacles. Figure 3.7 shows a top view of the maze arena, providing the robot with lots of walls and an opportunity for the robot to exhibit behaviours such as following a wall. Figure 3.8 shows a top view of the circular arena with black tracks on the ground providing the robot with an opportunity to exhibit behaviour such as following a track.

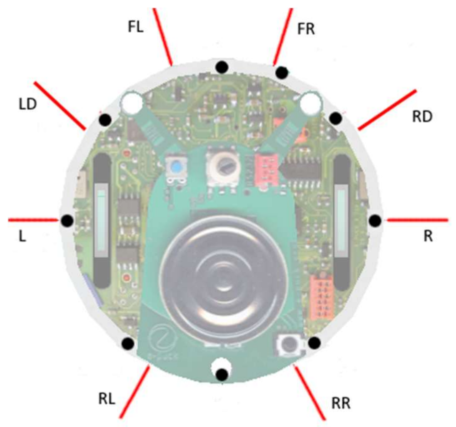


Figure 3.5: Top view of e-puck with labelled proximity sensors. Red lines show the direction in which the proximity is detected.

State Vector:

$[\omega^L \ \omega^R \ p^{FR} \ p^{RD} \ p^R \ p^{RR} \ p^{RL} \ p^L \ p^{LD} \ p^{FL} \ g^L \ g^C \ g^R]$

Actions:

{
 1 - Turn Left,
 2 - Step Forward,
 3 - Turn Right
}

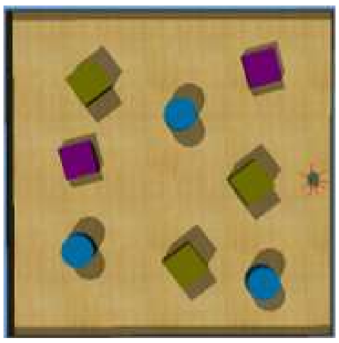


Figure 3.6: Top view of the arena with obstacles.

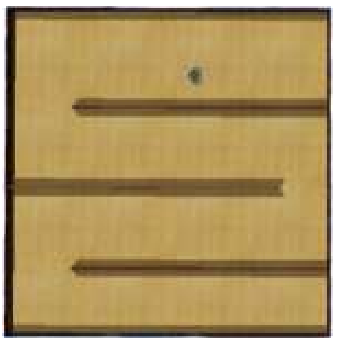


Figure 3.7: Top view of the maze arena.

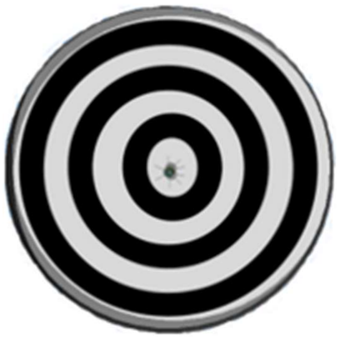


Figure 3.8: Top view of the circular arena with tracks.

Learning Algorithm

In the experiments in this chapter, a reinforcement learning algorithm called Dyna-Q (detailed in Chapter 2) was implemented using MATLAB. In the experiments, the potential tasks to be learned were considered 'maintenance' tasks, where the aim is to maintain a target state [32]. That is to say, the mobile robot aimed to maintain the desired state. That makes learning non-episodic. Thus, the concept of 'trial' was used to represent the start and end of an attempt. Each trial consisted of 50,000 steps, after which the trial was ended. The epsilon parameter of the epsilon greedy action selection was set to 0.1 with no decay for the experiments.

Chapter 4 details the maintenance task type and other types of tasks when categorised by how they are considered attained. In the experiments in this section, the following generic reinforcement learning reward function was used:

$$r(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \mathit{if\ the\ task\ state\ is\ maintained} \\ -0.5 & \mathit{otherwise} \end{cases} \tag{3.1}$$

where r is the reward received by the mobile robot in step s_t. Also, for the experiments, the learnability of a task was determined by calculating the average cumulative reward R for a configured number of trials. That configured number was 5, i.e. after five attempts, the task was deemed unlearnable if this cumulative reward was less than the threshold. As shown by the equation,

$$R < -(0.4 * number\_of\_steps\_per\_trial) \tag{3.2}$$

it was concluded that the system was unable to learn to maintain that task state, resulting in the task being removed from the list of potential tasks.

Task Generation Technique

The experiments in this chapter used the ART1 (detailed in Chapter 2) clustering algorithm as the task generation technique. For the experiments in this chapter, the vigilance parameter value was set to 0.3, resulting in relatively fewer categories. Once the data points have been categorized/clustered, the next step is to select a representative data point. In the experiments, for simplicity, the first data point in the category was selected as a representative data point, which was then considered a 'potential task'.

Consider a hypothetical state [1 1 0 0 0 0 0 0 1 1 0 0 0] that is selected as a potential task. If this task is treated as a maintenance task, it would mean that the e-puck has to maintain moving forward with the high sensor values of its *Left-Diagonal, Front-Left* sensors. Since the state vector does not comprise position or orientation attributes, this task can be attained at any location near the wall/obstacle. Figure 3.9 shows the top view of the maze arena with example locations of the e-puck in the maze arena. The mobile robots are shown in blue colour with red directional lines for the *Left-Diagonal, Front-Left* sensors. The task is considered maintained as long as the e-puck moves forward with high sensor values of its

Left-Diagonal, Front-Left sensors, i.e., to maintain that task e-puck is following a wall to its left.

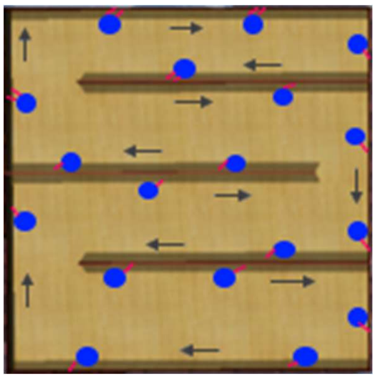


Figure 3.9: Example locations of the e-puck (shown in blue colour) in the maze arena for the state "move forward with the wall to its left".

3.4.3 Results and analysis

This section demonstrates the performance of each of the key modules of the architecture introduced in the previous section. The experiments in this chapter focus on the qualitative evaluation of the set of tasks generated and the architecture's continuous learning capacity. Chapter 4 introduces quantitative metrics to examine further these and other aspects of the architecture's performance.

Task Generation Performance

The architecture design allows any task generation mechanism to be used; however, the experiments in this chapter used ART1 based clustering algorithm to generate potential tasks. ART1 takes the experience gained by the mobile robot during the experience-gathering phase as the data points. Since there is no arena specific attribute in the agent's state space, this exploration phase, i.e. the experience gathering phase, aims not to explore the different parts of the arena but enables the agent to experience the different states in its state space. For example, it may be able to explore the black region on the floor or obstacle on one of its sides that are represented as different states in its state space. In this phase, the e-puck moves around randomly in the arena. That is done by making it follow an exploration policy. In this policy, the epsilon parameter of reinforcement learning's epsilon-greedy action selection strategy is set to 1, and the reward received is 0 for all the state action combinations. Webots simulates the dynamics of the environment, i.e. the collision behaviour. Thus, when the e-puck collides against an obstacle, it is not allowed to move forward anymore, and hence as per the exploration policy, it will randomly select a new action, landing in a new location in the arena and internally in a new state within its state space. This phase was carried out for 15,000 steps. The trajectory of the e-puck while in the experience-gathering phase is shown in Figure 3.10, Figure 3.11 and Figure 3.12.

The task generation mechanism takes the data points gathered during the exploration as the input and creates clusters as per the ART1 algorithm. Table 3.2, Table 3.3 and Table 3.4 show the cluster representatives for each of the three arenas. The first column of each table shows the task Id; the second shows the task attributes (robot's state)—these are the cluster representatives identified by ART1. The third column shows the manually provided description of those tasks. The architecture does not require that description but is done to provide an intuition of what those states mean. The fourth column shows whether the task previously existed in the knowledge management module's list or is a unique new task identified in this arena. The fifth column shows whether the task is valid or invalid.

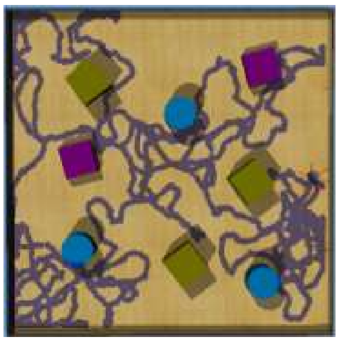


Figure 3.10: Trajectory, shown using blue line overlay of the e-puck randomly exploring the arena with obstacles. The states experienced during this exploration would be related to "being close to an obstacle", "being in an open space", to name a few.

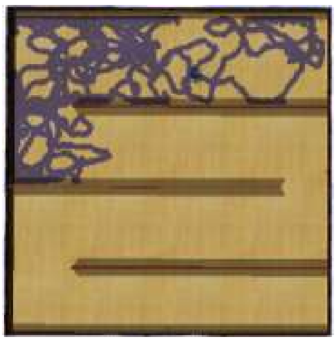


Figure 3.11: Trajectory, shown using blue line overlay of the e-puck randomly exploring the maze arena. The states experienced during this exploration would be related to "being close to a wall", "being in an open space", to name a few.

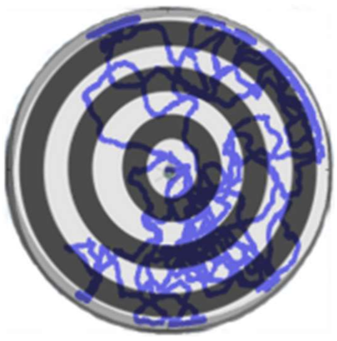


Figure 3.12: Trajectory, shown using the blue line overlay of the e-puck randomly exploring the circular arena with tracks. The states experienced during this exploration would be related to "being on a track", "not on a track", to name a few.

For the first arena, ten clusters are generated; for the second and the third arena, nine clusters are generated. The vigilance parameter for the ART1 algorithm influences the number of clusters generated by varying the similarity between the clusters. The output of the task generation provides a fair idea of the different cluster representatives, i.e. it provides an insight into the states experienced by the e-puck. From Table 3.2, it can be seen that the states experienced by the mobile robot are related to obstacles to its right or left while it is either moving forward or turning right or left. For example, one of the cluster representatives is T_{o9}. This state indicates that the mobile robot is turning right when there is an obstacle to its right. Table 3.3 shows the results for the maze arena (Figure 3.7), which can be considered similar to the arena with obstacles (Figure 3.6) since it has walls that also act as obstacles. That similarity is evident from the result, as many of the cluster representatives are the same as in Table 3.2. For example, one of the cluster representatives is T_{m7}. This state indicates that the mobile robot is moving forward with the wall to its right. Table 3.4, on the other hand, shows that the robot experiences the states that denote that it is partially or entirely over the black region on the ground. For example, one of the cluster representatives is T_{c3}, which indicates that the robot is moving forward while on a black track.

Table 3.2: Cluster representatives generated for the arena with obstacles.

Task Id	Task Attributes	Description of the task state	Unique task?	Is Task Valid?
T_{o1}	1 1 0 0 0 0 0 0 0 0 0 0 0	Moving forward, no obstacle anywhere nearby	Yes	Yes
T_{o2}	0 1 0 0 0 0 0 0 0 0 0 0 0	Turning left, no obstacle/wall nearby	Yes	Yes
T_{o3}	1 1 0 0 0 0 0 1 1 0 0 0 0	Moving forward, obstacle/wall on the left at the side	Yes	Yes
T_{o4}	1 0 1 0 0 0 0 0 1 1 0 0 0	Turning right, obstacle/wall on the left at the front	Yes	Yes
T_{o5}	0 1 1 1 0 0 0 0 0 1 0 0 0	Turning left, obstacle/wall on the right at the front and obstacle/wall on the left at the front	Yes	Yes
T_{o6}	1 0 1 1 1 0 0 0 0 0 0 0 0	Turning right, obstacle/wall on the right at the front and side	Yes	Yes
T_{o7}	1 0 0 0 0 0 0 1 1 0 1 1 1	Appears to be invalid sensor readings. Ground sensors should be 0 for this arena.	Yes	No
T_{o8}	1 1 0 0 0 0 1 1 0 0 0 0 0	Moving forward, obstacle/wall on the left at the back	Yes	Yes
T_{o9}	1 0 0 1 1 0 0 0 0 0 0 0 0	Turning right, obstacle/wall on the right at the side	Yes	Yes
T_{o10}	0 1 1 0 1 0 0 0 0 1 0 0 0	Turning left, obstacle/wall at the front and right at the side	Yes	Yes

Table 3.3: Cluster representatives generated for the maze arena.

Task Id	Task Attributes	Description of the task state	Unique task?	Is Task Valid?
T_{m1}	1 1 0 0 0 0 0 0 0 0 0 0 0	Moving forward, no wall anywhere nearby	No	Yes
T_{m2}	0 1 0 0 0 0 0 0 0 0 0 0 0	Turning left, no wall anywhere nearby	No	Yes
T_{m3}	1 1 1 0 0 0 0 0 0 1 1 0 0 0	Moving forward, a wall at the front and left	Yes	Yes
T_{m4}	1 1 0 0 0 0 0 0 1 1 0 0 0 0	Moving forward, the wall on the left at the side	No	Yes
T_{m5}	1 0 1 1 0 0 0 0 0 0 1 0 0 0	Turning right, a wall at the front and right	Yes	Yes
T_{m6}	1 1 1 1 0 0 0 0 0 0 0 0 0 0	Moving forward, a wall on the right at the front	Yes	Yes
T_{m7}	1 1 0 1 1 0 0 0 0 0 0 0 0 0	Moving forward, a wall on the right at the side	Yes	Yes
T_{m8}	1 0 0 0 0 0 0 0 0 0 0 1 1 1	Appears to be invalid sensor readings. Ground sensors should be 0 for this arena.	Yes	No
T_{m9}	1 1 1 1 0 0 0 0 1 1 1 0 0 0	Moving forward, a wall at both the front right and left.	Yes	Yes

Table 3.4: Cluster representatives generated for the circular arena with tracks.

Task Id	Task Attributes	Description of the task state	Unique task?	Is Task Valid
T_{c1}	1 0 0 0 0 0 0 0 0 0 0 0 0 0	Turning right, no wall anywhere nearby, and the robot is not on the black track	Yes	Yes
T_{c2}	0 1 0 0 0 0 0 0 0 0 0 0 0 0	Turning left, no wall anywhere nearby, and the robot is not on the black track	No	Yes
T_{c3}	1 1 0 0 0 0 0 0 0 0 0 1 1 1	Moving forward, no wall anywhere nearby, and the robot is on the black track	Yes	Yes
T_{c4}	0 1 0 0 0 0 0 0 0 0 0 1 1 1	Turning left, no wall anywhere nearby, and the robot is on the black track	Yes	Yes
T_{c5}	1 1 1 1 0 0 0 0 0 0 1 0 0 0	Moving forward, a wall at the front on the right and left	Yes	Yes
T_{c6}	1 1 0 1 1 0 0 0 0 0 0 0 0 0	Moving forward, a wall on the right at the side and the robot is not on the black track	No	Yes
T_{c7}	1 1 0 0 0 0 0 0 0 1 1 0 0 0	Moving forward, a wall on the left at the front and the robot is not on the black track	Yes	Yes
T_{c8}	1 1 0 0 0 0 0 0 1 1 0 0 0 0	Moving forward, a wall on the left at the side and the robot is not on the black track	No	Yes
T_{c9}	1 1 0 0 0 0 0 0 1 1 0 1 1 1	Appears to be invalid sensor readings. There is no track near the wall.	Yes	No

Scanning through the manually provided description of the tasks, i.e. cluster representatives, most of the tasks appear to be valid. However, a few appear to be invalid cluster representatives, for example, state T_{o7} in Table 3.2 and the state T_{m8} in Table 3.3. There are no dark coloured regions on the floor in either of those arenas. Examining the raw data points from the exploration phase also shows the presence of those states. The only explainable reason for those data points and subsequently identified cluster representatives is the sensor noise. However, this also shows that the task generation mechanism is correctly identifying the uniqueness of the clusters. The knowledge management module will prune such invalid tasks.

Knowledge Management Performance

In the experiment, the knowledge management module maintains a list of cluster representatives identified by the task generation module and their status, indicating whether they are learned or not. These cluster representatives are termed as ‘potential tasks’, i.e. unique states that the e-puck has experienced. The knowledge management module also stores the reinforcement learning policy, i.e. the Q-table as the learned knowledge. Q-table can be persisted as a text file and stored. They can be recalled back into memory when required. Comparing the results shown in Table 3.2, Table 3.3 and Table 3.4, it can be seen that several ‘potential tasks’ are common to the different arenas, such as state T_{o1}, T_{m1} and T_{c1}. This state means that the e-puck is moving forward while in an open space and not on the black region on the floor. The knowledge management module is responsible for keeping track of such similarities, as indicated by the third column of Table 3.2, Table 3.3 and Table 3.4. Initially, for the first arena, the task generation module will identify a large number of tasks that are added to the list of tasks to be learned by the knowledge management module. However, the number of tasks added will be fewer for the arenas that are explored subsequently, as only the unique tasks are added to the list. Once a task is learned, it is marked as learned, and its policy is stored in the repository. If a task cannot be learned, it is marked as such and left on the list for later attempts. The task cannot be learned because it is too difficult at that point in time or because it is an invalid task. The learning was attempted up to five times, after which such tasks were removed from the list.

Learning Performance

The design of the proposed architecture permits the usage of any reinforcement learning algorithm. However, since the experiments in this chapter aim to focus on the validity of the proposed architecture, the task representation uses discrete state and action space, and the Dyna-Q reinforcement learning algorithm was used. Once the ‘potential task’ generated by the task generation module was learned, the policy, i.e. the learned knowledge, was stored in the knowledge management module. The learning module was then assigned another task to learn, and the cycle continued.

In the experiments, the tasks were considered ‘maintenance’ tasks, i.e. the aim of the agent was to maintain the task state, and the reward detailed by Equation 3.1 was used by the algorithm. Using the intuition provided by the description column of Table 3.2, Table 3.3 and Table 3.4, five tasks were selected for each arena, the results for which are shown in Figure 3.13. These tasks were selected as follows: one task of particular interest, one invalid task and the remaining three were randomly chosen. The five tasks for each of the arenas are shown in the legend of the figure.

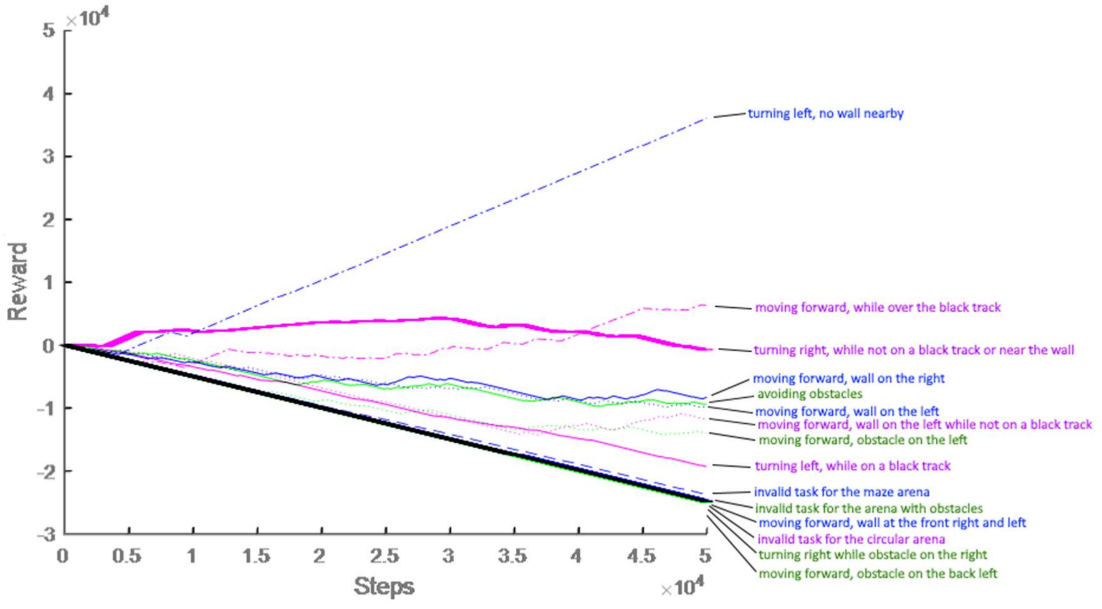


Figure 3.13: Reward gained when learning the selected tasks. Tasks for the arena with obstacles are shown in green, tasks for the maze arena are shown in blue, and the tasks for the circular arena are shown in purple.

Figure 3.13 shows that the invalid tasks are not learned. The cumulative reward for them is -25000, the lowest that can be in the experiments. For the other tasks that are valid, the learning performance depends on the opportunity that the e-puck gets to learn a particular task. Results show that some tasks are learned, others are not. In this experiment, the aim was to perform qualitative analysis; hence the experiment for each task was run for only 50,000 steps. By very nature, in reinforcement learning, the learning requires that the agent tries out different actions in a particular state. For this, the agent has to find itself in that state to try out other available actions. Some states are easy to regain the maintenance attempt whereas others are not, thus the difference in the opportunity to learn certain tasks. Chapter 4 introduces metrics to measure this, and an experiment in Chapter 6 shows that the learning performance can be better in specially constructed environments as it provides a better learning opportunity. The task T_{o2} (turning left, no wall nearby) in the maze arena appears to be easy to learn. The e-puck finds plenty of opportunities (open area) and keeps turning left. For the tasks that appear to be difficult to learn, in the experiment, if the cumulative reward was less than the threshold as shown by Equation 3.2, it was considered a failed learning attempt and such task was added to a list to be reattempted. After five attempts, the task was considered unlearnable and removed from the task list.

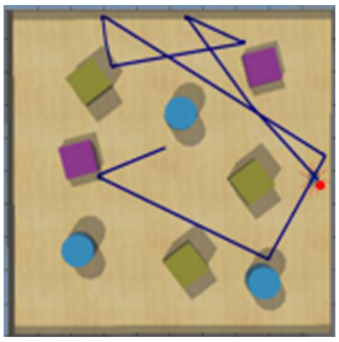


Figure 3.14: Trajectory of e-puck (shown using blue line overlay) avoiding obstacles/walls in the arena with randomly scattered obstacles. The starting location is shown with the red dot.

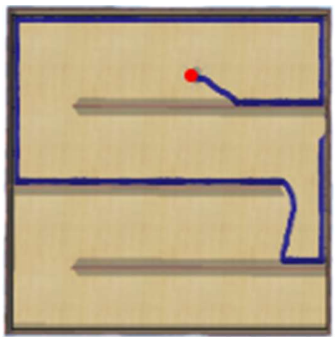


Figure 3.15: Trajectory of e-puck (shown using blue line overlay) following the wall to its right in the maze arena. The starting location is shown with the red dot.

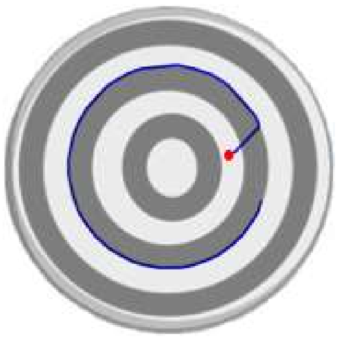


Figure 3.16: Trajectory of e-puck (shown using the blue line overlay) following the black track in the circular arena. The starting location is shown with the red dot.

Regarding the tasks of particular interest for each of the arenas, the learning was continued for one million steps. For the arena with obstacles, the task state of particular interest is T_{o1}. To maintain this state, the e-puck has to maintain zero sensor values of all its proximity sensors while moving forward, i.e. avoid obstacles. In learning this task, the e-puck exhibits the behaviour of avoiding obstacles. Figure 3.14 shows the learned behaviour that the e-puck is avoiding obstacles. For the maze arena, the task state of particular interest is T_{m6}. To maintain this state, the e-puck has to maintain high sensor values of its *Front-Right, Right-Diagonal* proximity sensors and zero value for the rest of its proximity sensors while moving forward. In learning this task, the e-puck exhibits the behaviour of following the wall to its right. Figure 3.15 shows learned behaviour. For the circular arena with tracks, the task state of particular interest is T_{c3}. To maintain this state, the e-puck has to maintain zero sensor values of all its proximity sensors and high sensor values of its ground sensors while moving forward. In learning this task, the e-puck exhibits the behaviour of following the track. Figure 3.16 shows the learned behaviour of the e-puck following the black track.

Continuous Learning and Overall Architecture Performance

The continuous learning cycle starts with the exploration phase, where the e-puck moves around randomly in its arena. The states experienced during this phase are used by the task generation module to generate potential tasks. Those tasks are then learned, and the knowledge is stored in the knowledge management module. The robot is then placed in another arena, and the cycle continues. Figure 3.17 summarises this continuous learning. When the e-puck is placed in a new arena, it forms new tasks and learns those tasks, i.e. when placed in a new environment, it autonomously discovers new tasks specific to that environment and then learns to attain those tasks, thus continuously improving its knowledge of its environment.

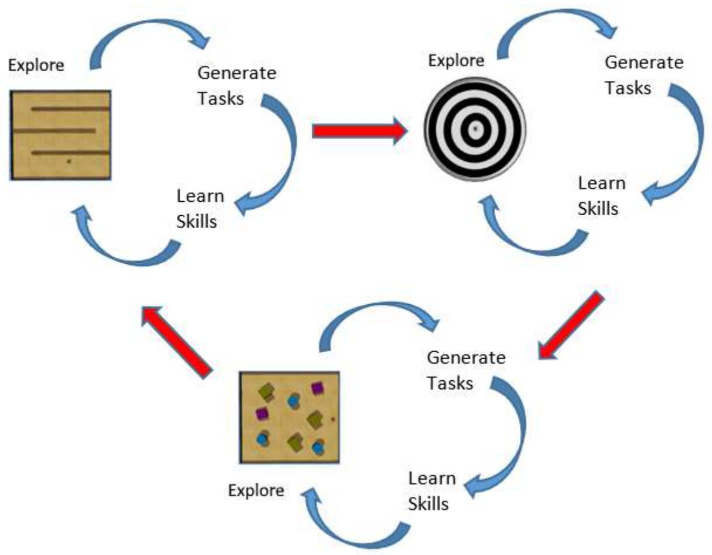


Figure 3.17: Continuous Learning cycle of generating tasks and learning skills in different arenas. With this open-ended continuous learning cycle, the e-puck discovers a unique set of tasks in each arena and then learns skills to accomplish those tasks, thus increasing its overall knowledge base.

Again consider a few examples from each of the three arenas. As seen in Table 3.2, when the e-puck is in the arena with obstacles, the potential tasks generated are related to the presence of an obstacle in its proximity (for example, T_{o8}) or that it is in an open space (state T_{o1}). When learning tasks such as T_{o1}, the e-puck exhibits behaviour akin to avoiding obstacles. Next, when the e-puck is placed in the maze arena, as shown in Table 3.3, it forms tasks related to being beside a wall (for example, T_{m7}). When learning to maintain such a task state, the e-puck exhibits behaviour akin to following the wall on its right. Then, when the arena is changed again to the circular arena with tracks, as shown in Table 3.4, it forms tasks related to being over a track (for example, T_{c3} or T_{c4}). In learning to maintain such task states, the e-puck exhibits behaviour akin to following the track.

Thus, the e-puck goes from having no knowledge of its environment or even its own state space to exhibiting identifiable behaviours. At the start and when the environment changes, the e-puck self-generates the tasks specific to that environment and learns skills to accomplish those tasks, thus, continuously improving its overall knowledge. That shows the validity of the proposed continuous learning architecture.

3.5 Summary

This chapter proposed the Modular Continuous Learning Architecture—an agent architecture with a 'Task Generation Module' that enables the agent to decide what to learn, a 'Knowledge Management Module' which is a skills repository and a 'Learning Module' implemented using reinforcement learning. The task generation, learning, storing, and recalling of the skill continues in a cycle, thus continuously improving the system's overall capability without external intervention. Mobile robot experiments were run in a dynamically changing environment to demonstrate how the agent can switch from learning to exploring and continuing that in an open-ended manner, with basic definitions for the key components: tasks, skills and rewards. When the Task Generation Module is implemented using novelty-based or curiosity-based motivation or the Learning Module is implemented using intrinsic motivation based reward, it creates a 'motivated open-ended continuous learning' architecture. The literature review showed that the focus of the motivated reinforcement learning agent architectures was on generating task-independent reward functions and the focus of goal-oriented autonomous agent architectures was on generating tasks to self-direct the learning, both of which lacked lifelong learning capability. This new architecture fulfils that gap in the literature.

The architecture presented in this chapter provides the foundation for investigating open-ended continuous reinforcement learning by mobile robots in the rest of this thesis. The next chapter will examine the types of tasks and the design of generic rewards for different task types in the context of the proposed architecture. Later chapters will examine compound task generation (Chapter 5) and skill composition (Chapter 6).

CHAPTER 4 REWARD DESIGN FOR AUTONOMOUS LEARNING

Parts of this chapter have been published in: P. Dhakan, K. E. Merrick, I. Rano, and N. Siddique, "Intrinsic rewards for maintenance, approach, avoidance, and achievement goal types," Frontiers in Neurorobotics, vol. 12. October. 2018.

4.1 Introduction

The previous chapter showed the modules that make up the Modular Continuous Learning Architecture. Those modules form essential components for the agent architecture's open-ended and continuous learning aspects. Another crucial aspect that was mentioned in Chapter 1 was autonomous learning, which is the main reason for using reinforcement learning in architecture. In reinforcement learning, the agent is not provided labelled data. It is not told what the positive and negative samples are. Neither is there a control program consisting of if-else conditions instructing the agent what to do. Instead, the agent must figure out what action it should take in which state. That mapping from state to action is called policy, and the learning is driven using 'reinforcement', which can be a negative or a positive reward. Typically, the reward in single task reinforcement learning is a handcrafted function that may require significant domain knowledge in many cases. For agent autonomy, it is essential that the rewards are task-independent. The literature review shows that either intrinsic motivation or reward shaping can be used as a task-independent reward function. However, another way of designing the task-independent reward function is basing it on the type of the task. That is what will be discussed in this chapter.

Tasks, also interchangeably referred to as goals, have been among the main research areas of the Beliefs, Desires, Intentions community [71] and the agent community [72]. As seen in Chapter 2, a categorisation based on how the tasks are attained leads to 'maintenance', 'achievement', 'approach' and 'avoidance'. In a reinforcement learning problem formulation, albeit somewhat unknowingly, this task categorisation is already considered. Take, for instance, the cart-pole benchmark problem, which is a maintenance task. A maze navigation problem is an achievement task. Similarly, problems solved with positive reward have properties of approach task, and the problems solved using negative reward have properties of avoidance task. Thus, the concept of using task types for generating a reward seems promising. Encouraged by that, this chapter proposes a task-independent reward function for different types of tasks.

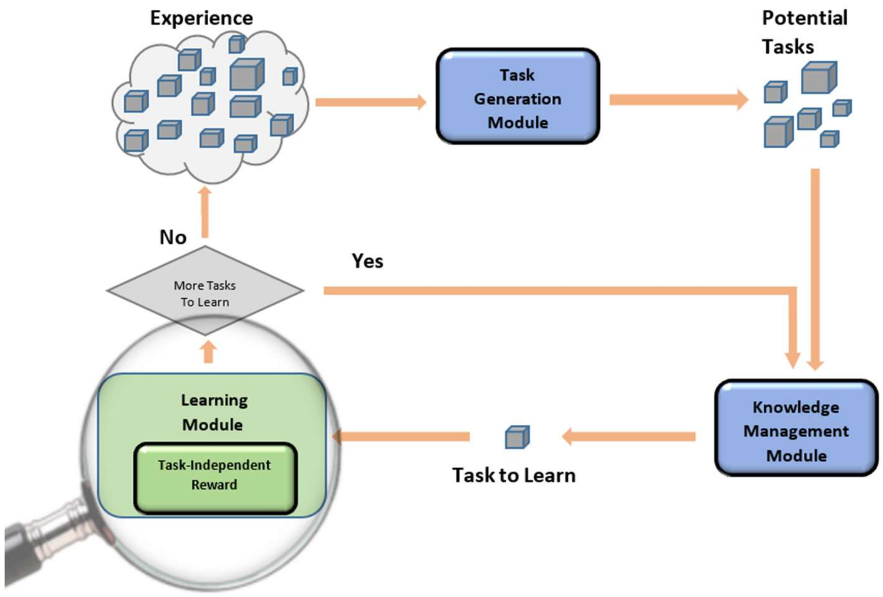


Figure 4.1: Modular Continuous Learning Architecture revisited. Task-independent reward design, the focus of this chapter, is the contribution related to the Learning Module of the architecture.

This chapter will propose reward functions for achievement, approach, avoidance and maintenance task types. Also, tasks can be 'primitive', i.e. elementary tasks that logically cannot be broken down into subparts and 'compound' that are a combination of the primitive tasks. This chapter will further explore the possibility of extending the proposed reward functions to be used with the compound tasks. It will also propose metrics to measure the performance of the agent. Using simulated e-puck based experiments, this chapter will demonstrate the use of the proposed reward functions, i.e. the robot will learn the different types of tasks. One of the experiments will also show a hand-coded example of how a compound task can be broken into sub-tasks, which could be treated as one of the aforementioned types. Then using the proposed reward functions for those sub-tasks, the experiment will find a solution for the compound task.

Figure 4.1 is the Modular Continuous Learning Architecture, as detailed in Chapter 3. The 'Task Learning Module', shown in green, uses reinforcement learning. The task-independent reward function design, the focus of this chapter, is a research contribution related to that learning module. The rest of this chapter is organised as follows: Section 4.2 will review the literature on task-independent reward design. Section 4.3 proposes the reward functions for different types of tasks. Section 4.4 proposes the metrics to measure the agent's performance. Section 4.5 will detail the setup of the experiments and discuss the results. Finally, Section 4.6 will then provide the concluding remarks.

4.2 Task-Independent Reward Design

Reinforcement Learning, where an agent learns by interacting with its surroundings, is most suitable for autonomous learning. The learning is guided by reinforcement, commonly known as 'reward'. The reinforcement learning agent aims to maximise the cumulative reward and, in doing so, find the optimal mapping between states and actions, i.e. learn the task. Thus, reward design is most crucial in reinforcement learning. 'Reward engineering' is one of the active research areas. That research focuses on the principles of construction of reward that empowers efficient learning [73]. Typically, designing a good reward function requires task-dependent knowledge. However, if the reward is task-dependent, it hinders autonomy. Thus, this section will review the literature on a specific aspect of reward engineering, the one that is concerned with task-independent reward design.

Consider the following example of a predefined value assignment reward function:

$$r(s_t, a_t, s_{t+1}) = \begin{cases} 1 & if\ a\ particular\ state\ is\ reached \\ 0 & otherwise \end{cases} \tag{4.1}$$

where $r(s_t)$ is the per-step reward at time step *t*. Such reward is suitable for certain tasks; however, consider a benchmark cart-pole (or the similar inverted pendulum) example used in reinforcement learning research. In this problem, there is a cart that moves along a track with a pole attached. The agent's task is to balance that pole by moving the cart to the right or left along the track. The following equation gives the reward function where u_t^1 and u_t^2 are its state attributes, the position of the cart and the pole's angle with respect to the cart. G is the goal state with attributes g^1 and g^2, the desired position and angle.

$$r(s_t, a_t, s_{t+1}) = -c2 * (g^1 - u_t^1)^2 - c3 * (g^2 - u_t^2)^2 \tag{4.2}$$

As can be seen from the reward function, such reward design requires significant task-dependent knowledge. In essence, the control logic is encoded in the reward function, instructing the agent what action to take when in a particular situation. Consider another example of a ball paddling game. The game consists of a ball attached to a paddle by an elastic string. The aim is to bounce the ball on the paddle. To design a good reward function, one has to consider not just that the end task is accomplished but also how the task is accomplished. Thus, the reward function should consist of a condition for bouncing the ball and a condition for bouncing it above the paddle and not hitting it towards the ground, again illustrating the need for task-dependent knowledge to design a reward function. If the design is incorrect, the agent will learn an incorrect behaviour [74]. Even if the design is correct, the agent might find an alternative way to gain maximum cumulative reward, i.e. 'hack' the reward [74], resulting in the right behaviour but achieved in an unforeseen way. Thus, a good reward design usually consists of a primary reward responsible for guiding the agent towards the goal and a secondary reward that guides the agent in the way the goal is to be achieved.

Also, for some domains, it is only possible to design a 'sparse reward'. This is where the reward is assigned to a small proportion of situations making the learning difficult since, in such cases, the reinforcement learning agent gets very little feedback for its actions. Alternatives proposed in the literature include 'hallucinating' positive rewards [75]. Another solution is to use imitation learning [33] [76] and inverse reinforcement learning [77] [78] that provide a near-optimal policy, and the reward can then be derived from these human demonstrations. However, this does not allow the fully autonomous development of the agent. Dewey [73] concluded that to create an autonomous agent, one cannot use handcrafted rewards, and that is a considerable challenge since the task-independent design of reward resulting in desired behaviours is complex. While earlier, the focus of the reinforcement learning research was on efficient learning of an arbitrary given task, recent research has recognised that the design of a reward function can either restrict or facilitate autonomy. Also that the reward function can enable open-ended learning, allowing reinforcement learning to progress beyond single-task learning. The following few subsections review work that focuses on this area.

4.2.1 Intrinsic motivation

Although the concept of novelty [79] and curiosity [80] has been used with reinforcement learning for decades, this usage was to find novel tasks to learn or assist with exploring the state-space. Singh et al. [45] introduced the idea of reward modelled using the psychological concept of motivation. Motivation is defined as a reason to perform a task and is broadly classified as either intrinsic or extrinsic [81]. When a task is carried out for internal satisfaction without any external influencing factor, it is said to be carried out due to 'intrinsic motivation'. When a reward signal is generated internally within an agent using an inherent attribute without an external influence, it is classified as an intrinsic reward [45]. It can depend on the state components from the agent's internal environment in addition to the components from its external environment and is task-independent [82]. It is, in short, an agent's perception of the scalar reward and an example of an engineered reward leading to open-ended learning [83] [84].

The computation of the reward can be based on experienced states, specific events, or actions. The central aspect is that it is independent of the previous task-specific knowledge. The motivation signal may lead to learning a specific skill of no immediate benefit but could be beneficial later [45]. An agent may generate this signal because a task is inherently 'motivating', which in turn results in further exploration of its environment or acquisition of the skill and is composed of the agent's perception along with components from its external environment.

$$r = r_e + r_i \tag{4.3}$$

Intrinsic motivation, i.e. the 'motivation function', is a reward model that leads to task-oriented performance. Equation (4.3) shows that the reward r is a summation of the extrinsic or hand-coded task-specific reward r_e and the intrinsic reward r_i. However, the intrinsic reward may be used along with or instead of a task-specific reward signal. When used along with the task-specific reward, the agent achieves more adaptive learning. When used instead of the task-specific reward signal, it results in a true task-independent learner since it reduces the handcrafting of the task-specific reward [44]. Also, alternatively, the intrinsic reward can be gradually decreased or increased as required. When intrinsic motivation is used in reward design, it serves the following two purposes: (i) state-space exploration and (ii) controllability, i.e. to provide an internally generated positive or negative feedback to manipulate the agent's behaviour. In this section, the review is restricted to the latter form of intrinsic motivation. Section 2.4 reviews the former use of intrinsic motivation.

Intrinsic motivation is categorised by Oudeyer and Kaplan [85] into the following two categories: knowledge-based and competence-based. The signal is considered knowledge-based motivation if it is based on an internally generated prediction error [86]. This error is generated based on the comparison between what actually happens versus the agent's expectation of what is supposed to happen when a particular action is executed and is also sometimes referred to as 'surprise' [87]. On the other hand, novelty identifies new patterns that have never been seen before [79] [88]. Intrinsic reward is generated only by a novel or unexpected event. The system compares the predicted next state to the actual next state, and if the prediction is incorrect, a novelty signal is generated. Competence refers to an organism's ability to interact with the environment and the development of its ability to change it in specific ways. It is the sense of mastery that the organism has for a particular skill. The central concept for a competence-based model is an appropriate level of learning a challenge; that is, the activity is at a correct level of learnability given the agent's current level of mastery of that skill [89]. The model gauges the agent's competence in achieving the self-determined goals. The signal is considered competence-based motivation if it is generated based on a progress indicator metric [86]. Such motivation generates a maximum signal when the task's difficulty level matches the agent's mastery of the skill required to accomplish that task.

The following is a small subset of examples of how intrinsic motivation is derived. Oudeyer et al. [54] use the distance measure between the terminal state when the goal-reaching attempt is finished and the actual goal state as a measure of competence to derive an intrinsic reward. Stout and Barto [90] used the expected pseudo return of the options to derive the competence-based reward. Bonarini et al. [91] use a 'level of interest' in visiting the states as an intrinsic measure to generate subgoals. They propose that the states that are difficult to reach during random exploration, once reached and easily exited, generate a high level of interest. Sequeira et al. [92] propose using an agent's emotions such as surprise, in control, and situation pleasantness as intrinsic motivation. Temel et al. [93] and

Grzyb et al. [94] use the frustration of the robot in grasping the objects as a measure of competence progress. They showed that low or high frustration results in exploitative behaviour, whereas an optimal level of frustration results in explorative behaviour. Along similar lines, Ma et al. [95] have an optimal level of challenge between the task's difficulty level and one's competency is essential to maintain a higher level of intrinsic motivation. Motivation has been examined in a deep reinforcement learning setting as well. Kulkarni et al. [96] present a hierarchical DQN framework where motivation is used to identify the goals for the agent in a data-efficient manner. Bellemare et al. [97] use intrinsic motivation generated using the count-based exploration method to measure the learning progress and demonstrate its advantage on Atari 2600 games and how this leads to better state space exploration.

Santucci et al. [98] compare the different knowledge-based and competence-based intrinsic motivations. The motivation generated for the state predictor, the state-action predictor and the task predictor is compared using prediction error and prediction error improvement measures. Their results show that just knowledge-based motivation is insufficient for an agent to learn skills for multiple tasks, whereas some but not all types of competence-based motivation are sufficient. Whether knowledge-based or competence-based, information regarding what constitutes prediction error or mastery level competency must be defined. However, they can be based on the attributes from the agent's internal state-space or components from its external environment but independent of task-specific factors. The following subsection reviews the literature on reward shaping.

4.2.2 Reward shaping

Another technique that generates task-independent reward is a concept called 'shaping' [99] [100] [101]. In reward shaping, one starts with a basic reward function that is then shaped either statically or dynamically. Thus, shaping can guide the learning process by favouring certain behaviours [102] and accelerating learning [103]. In the literature related to reward shaping, the reward is seen as a programmer's bias. The native task reward, the initial bias, is shaped by providing a positive or negative artificial increment to encourage or discourage behaviours. In essence, 'shaping' is a mathematical representation of a bias that will establish a preference for action [102]. Thus, shaping, in a way, is similar to the concept of intrinsic motivation seen in the previous subsection and can be called a precursor to it. The shaped reward $\tilde{r}$ can be represented using the following equation where r is the native task reward, f is the shaping function, (s_t, a_t, s_{t+1}) are the state, action and next state resulting from taking action a_t.

$$\tilde{r}(s_t, a_t, s_{t+1}) = r(s_t, a_t, s_{t+1}) + f(s_t, a_t, s_{t+1}) \tag{4.4}$$

In the case of static shaping, the reward function is modified using predetermined criteria, often hand-coded, i.e. it does not vary with experience. However, since this predetermined criterion is often hand-coded, it requires significant domain knowledge and external manual intervention. However, this concept can be further extended by modifying the reward dynamically. One approach is to derive the dynamic shaping function using the agent's initial experiences with the environment [103]. Another approach is to shape the reward based on the progress indicator [104], where the reward is shaped based on the evaluation of the progress in attaining a task.

Although shaping is said to be a powerful technique, it is also acknowledged that a poorly shaped reward function might cause the learning to converge on a non-optimal solution [105]. Also, designing a good shaping function that reduces learning time requires a task or environment-specific knowledge [106]. Thus, reward shaping is not the type of task-independent reward function that can be used for autonomous learning and is not explored further in this thesis. In saying that, the modular nature of the architecture proposed in Chapter 3 allows the 'Learning Module' to be implemented using reward shaping.

4.2.3 Gap in the state-of-the-art

Intrinsic motivation is the most commonly used approach to generate a task-independent reward function for reinforcement learning. While intrinsic motivation can be used instead of or along with other reward functions and leads to learning autonomy for an agent, it requires additional albeit general information. That information may or may not be task-independent in all cases. In the case of knowledge-based motivation, which is derived from the prediction error, the information on what constitutes novelty is required. In the case of competence-based motivation, which is derived from the difference between the mastery level competence and the current competence level, information on what constitutes mastery level competency is required. The other technique for generating task-independent reward found in the literature is using reward shaping. That, too, requires an additional input of either determining criteria for the static shaping update or determining what constitutes a dynamic update to the reward function. Thus, none of the existing task-independent techniques enables full autonomy.

Early work on reinforcement learning focused on generating reward, using intrinsic motivation directly from the environment. However, in many cases, especially where the rewards are sparse, research has acknowledged the benefits of generating subtasks, i.e. transitional milestones, to direct the learning [12] [38]. This approach enables the agent to break down a monolithic task, and the intrinsic reward for the subtasks increases the reward density for the overarching task. The agent can exhibit behaviours such as achieving, avoiding or maintaining these subtasks. That leads to an interesting alternative approach where the reward function is generated based on different types of tasks. That raises the question, how does one design a module to generate task-independent intrinsic reward functions for different types of tasks? The following section will aim to answer that question.

4.3 Reward Functions based on the Type of the Task

In reinforcement learning, the learning process is guided by reward. Typically the reward is hand-designed and often task-dependent. However, it is not always possible, especially in dynamic environments, to know upfront which tasks the agent should learn, making it challenging to design the reward function. This chapter proposes reward functions based on the type of task, which is the categorisation of tasks based on the functional aspect of how the task is considered to be attained. The common types based on this categorisation are achievement, maintenance, avoidance, approach, optimisation, test, query, and cease type [32]. The 'optimisation' type means maximising or minimising a particular value. The 'test' type determines if a particular condition holds. The 'query' type is used for information retrieval, and the 'cease' type is the opposite of the 'achieve' task type. These task types are more appropriate for data-driven agents than robotics applications [38]. The scope of this chapter is limited to the types of tasks whose "attainment" can be verified in the context of reinforcement learning. Hence optimisation, test, query, and cease task types are not considered further in this chapter. The remaining task types, achievement, maintenance, avoidance, and approach, as seen earlier, are already considered in the reinforcement learning problem formulation. Based on that criteria, this chapter proposes reward functions for those types. The basis of the reward design starts with the generic reward function, as shown in Equation (4.5).

$$r(s_t, a_t, s_{t+1}) = \begin{cases} 1 & \textit{when the task state is reached} \\ 1 - \varepsilon & \textit{in all other states} \end{cases} \tag{4.5}$$

where r is the per time step reward when action a_t is taken in the state s_t. ε is a non-negative constant; typically, its value would be greater than 1 so that there is a per step penalty which incentivises the agent to find an optimal solution. The rest of this section defines different representations of tasks and representations of the meaning of "reached" or "attained".

4.3.1 Reward function for a maintenance task

A type of task where the aim is to preserve the desired state is classed as a maintenance task. Thus, for a maintenance task, the distance between the current state and the desired state is monitored, and if that distance increases beyond a set threshold, the agent aims to re-establish the desired state. To do that, the agent's action selection should take into account both the triggering and constraining conditions of the task [107]. For example, consider a vacuum cleaning robot where the task is to remove dirt and dust from an area supposed to be "maintained clean". Examples of the mobile robot include following a wall or staying within a track. Other maintenance tasks are air conditioners maintaining a specific temperature in the room or cruise control maintaining a certain driving speed. Maintaining a task state can be never-ending. Thus, compared to "typical" reinforcement learning tasks, the maintenance tasks are non-episodic. Therefore, new metrics are required to measure the agent's performance. Those metrics are detailed in Section 4.4.

To represent the reward design, consider that the desired state is denoted by G and the agent's state at the time step t is denoted by s_t and that it takes action a_t in that state. Also, consider that $d(.)$ is a measure of distance and ρ is a permissible threshold distance. Thus, if the distance between the current and desired state is below the threshold, the aim can be considered fulfilled. The reward function is as shown in Equation (4.6):

$$r(s_t, a_t, s_{t+1}) = \begin{cases} \sigma & if\ d(s_t, G) < \rho \\ \varphi & otherwise \end{cases} \tag{4.6}$$

In the equation, when the aim is attained, the reward is σ and φ otherwise. Generally $\sigma > 0$, i.e. a positive reward, but there is no particular recommendation regarding the maximum value of σ. φ is less than σ in order to incentivise the agent to find an optimal solution.

4.3.2 Reward function for an approach task

A type of task where the aim is to get closer to the desired state is classed as an 'approach' task. Compared to the maintenance task detailed in the previous subsection, the difference lies in how the task is fulfilled. There is a 'desired state' in both cases, and the agent's aim is related to that state. For a maintenance task, as long as the distance between the current state and the desired state remains under a certain threshold, it is considered fulfilled. For an approach type task, it is considered fulfilled as long as the distance between the current state and the desired state is decreasing [108]. The approach attempt is said to have ended once that distance is shorter than a certain threshold, thus making the approach task transient in nature. The task is classified as an approach type task as long as the distance between the current and desired states is getting shorter. However, once that distance is shorter than the threshold, the task is no longer considered 'approaching'. For example, consider a vacuum cleaning robot starting from the middle of a room with the aim/task of following the wall. For that robot, the initial task would be to approach the wall and then start the next task of maintaining a set distance from the wall while following the wall.

To represent the reward design, consider that the desired state is denoted by G, and the agent's state at the time step t is denoted by s_t and that it takes action a_t in that state. Also, consider that $d(.)$ is a measure of distance and ρ is a permissible threshold distance. The approaching attempt is measured by comparing the distance between the current s_t and desired state G with the state at the previous time step s_{t-1} and the desired state G. The equation also has a second condition which ensures that while the agent is approaching the desired state, the distance remains more than the defined threshold ρ. That is so that the fulfilment remains an "approach" endeavour and not "approach and achieve". The first condition in the equation is also the progress indicator and can be used on its own. Such progress indicators can be used to represent an agent's competence and derive a competence specific intrinsic motivation signal. The reward function is as shown in Equation (4.7):

$$r(s_t, a_t, s_{t+1}) = \begin{cases} \sigma & if\ d(s_t, G) < d(s_{t-1}, G)\ \ and\ \ d(s_t, G) > \rho \\ \varphi & otherwise \end{cases} \tag{4.7}$$

In the equation, when the aim is attained, the reward is σ and φ otherwise. Also, generally σ > 0, i.e. a positive reward and φ is less than σ in order to incentivise the agent to find an optimal solution.

4.3.3 Reward function for an avoidance task

The type of task where the aim is to remain away from the desired state is classed as an avoidance task. It is the opposite of the approach task detailed in the previous subsection. As the name suggests, avoidance is a behaviour where an agent stays away from a particular state or an object [108]. The task is considered fulfilled as long as the agent stays away from the state it should avoid, making this type of task transient in nature. For example, consider a robotic vacuum cleaner cleaning rooms while avoiding obstacles or an autonomous lawnmower cutting grass while avoiding obstacles.

To represent the reward design, consider that the desired state is denoted by G, and the agent's state at the time step t is denoted by s_t and that it takes action a_t in that state. Also, consider that $d(.)$ is a measure of distance and ρ is a permissible threshold distance. The equation has two conditions. The first measures the avoidance attempt by comparing the distance between the current s_t and desired state G with the state at the previous time step s_{t-1} and the desired state G. The second condition ensures that while the agent is staying away from the desired state, the distance remains more than the defined threshold ρ. It is a more intuitive way to represent the avoidance attempt. Either of the two expressions can be used on their own as well. The reward function is as shown in Equation (4.8):

$$r(s_t, a_t, s_{t+1}) = \begin{cases} \sigma & if\ d(s_t, G) > d(s_{t-1}, G)\ and\ d(s_{t-1}, G) > \rho \\ \varphi & otherwise \end{cases} \tag{4.8}$$

As is the case for the maintenance and the approach reward functions, when the aim is attained, the reward is σ and φ otherwise. Also, generally σ > 0, i.e. a positive reward and φ is less than σ in order to incentivise the agent to find an optimal solution.

4.3.4 Reward function for an achievement task

A type of task where the aim is to attain the desired state is classed as an achievement task [109]. The task is considered fulfilled when that desired state is reached. That makes the learning episodic in nature, i.e. a starting or an initial state of an agent and an end state. In the case of reinforcement learning, the learning process can start at the starting state and then end when the desired state is reached. That starting state can be different for each episode to make the learning more robust. The concept of 'event' (detailed in Chapter 2) is used to represent an achievement task. An event describes the transition of states of the agent. An achievement task is considered fulfilled when that transition is accomplished. Examples of achievement tasks are a vacuum cleaning robot making its way out of the room once it is cleaned or finding its way through the room back to the charging station.

To represent the reward design, consider that the desired state is denoted by G, and the agent's state at the time step t is denoted by s_t and that it takes action a_t in that state. Also, consider that $d(.)$ is a measure of distance and ρ is a permissible threshold distance. The reward function is as shown in Equation (4.9):

$$r(s_t, a_t, s_{t+1}) = \begin{cases} \sigma & if\ d(E_t, G) < \rho \\ \varphi & otherwise \end{cases} \tag{4.9}$$

Similar to equations in the previous subsections, when the aim is attained, the reward is σ and φ otherwise. Also, generally σ > 0, i.e. a positive reward and φ is less than σ in order to incentivise the agent to find an optimal solution.

4.4 Metrology for Agent Performance

As seen in the previous section, tasks can be of various types when categorized based on their functional aspect. However, the reinforcement learning tasks are typically assumed to be of achievement type. Even the tasks that do not necessarily fit the definition of achievement task are treated as one. The task has a start state and the desired end state. A trial, a run or a rollout is considered episodic where the episode is considered to have ended when either (i) the end state is reached, (ii) a configured undesired state, such as a robot has fallen down, is reached, or (iii) the number of steps in the episode has exceeded the set limit. Typically, the agent's per episode aggregate reward is used as a metric. The agent is said to be performing well if this per episode reward increases or reaches a set maximum. Such a metric is not sufficient for all the task types detailed in the previous section. This section proposes metrics to measure the agent's performance for non-episodic tasks. For instance, maintenance tasks are non-ending; thus, the concept of an episode is not relevant. Thus, a metric such as the regain attempt, a critical measure for non-episodic tasks, is proposed. Another metric is the number of times the agent was not able to avoid the state that it was supposed to avoid. These metrics evaluate the reward functions for those task types and are measured over a fixed period *T* of the agent's life. The proposed metrics can be used in conjunction with the reinforcement learning's standard per-episode reward metric to provide additional insight into the agent's learning performance.

4.4.1 Number of times for which the non-episodic task is attained (M_1)

This metric is applicable to maintenance, approach and avoidance tasks. It counts how often the agent maintains/approaches/avoids the desired state for two or more consecutive steps during a period *P*. For maintenance, approach and avoidance tasks, the process of 'attaining' the desired state is never-ending, and the learning 'episode' is not ended at the first occurrence of the task being maintained/approached/avoided. That is to say that the learning is not stopped when the desired state is 'reached'; the agent's position is not reset back to a pre-determined or random initial state; instead, the training continues. The maintained state could be lost for such unending tasks, or the approaching/avoiding attempt is lost; however, since the process is never-ending, the reinforcement learning agent reattempts to maintain/approach/avoid. The metric M_1 counts the number of times the agent reattempts. Thus, it provides a measure of the agent's competence in regaining the approach/avoidance/maintenance task.

$$M_1 = \underset{t=2\ldots P}{count}(t) \ \ such\, that \ \ r_t = \sigma \ and \ r_{t-1} \neq \sigma \tag{4.10}$$

where depending on the type of the task, Equations (4.6), (4.7) or (4.8) provide r_t and r_{t-1}. In the equation, σ is used as an example positive reward for r_t and r_{t-1}.

4.4.2 The longest period of maintenance task (M_2)

This metric is also used for maintenance tasks and calculates the longest stretch in terms of time steps for which the task was maintained. Once the reinforcement learning agent has learned the skill, the expectation would be that this metric would be of higher value than the agent who has not learned or partially learned the skill. Thus, this metric indicates the agent's competence in maintaining the task, with longer stretches indicating better agent performance. However, a lower value of this metric could result from a lack of learning opportunities for the agent or an unsuitable environment. This metric requires keeping track (or calculated from experience) of the maintenance attempt segments (represented as an array *J*). The maintenance attempt segment starts when $r_t = \sigma \ and \ r_{t-1} \neq \sigma$ and ends when $r_t \neq \sigma \ and \ r_{t-1} = \sigma$ (σ is used in the equation as an example of a positive reward). M_2 is the longest segment in the array *J*.

$$M_2 = \underset{j=1\ldots J}{max}(length\ of\ maintenance\ segment\ j) \tag{4.11}$$

4.4.3 Number of times task not avoided (M_3)

This metric is applicable to avoidance tasks. For approach and avoidance tasks, the reinforcement learning agent may actually end up 'reaching' the state that it is approaching or avoiding. In such cases, the approaching or the avoiding attempt ends. For the avoidance type tasks, that 'reaching' is considered a failure. This metric measures how often such failure occurred, i.e. it counts how often the agent fails to avoid the task state over a fixed period P.

$$M_3 = \underset{t=1\ldots P}{count}(t) \ such\ that \ \ d(s_t, G) < \rho \tag{4.12}$$

where $d(.)$ is the distance between the current and the task state and ρ is the distance threshold.

4.5 Mobile Robot Experiments

Previous sections proposed the task-independent reward functions and metrics to evaluate an agent's performance. This section uses those metrics to verify the effectiveness of the reward functions to guide learning. The experiments in this section will use the e-puck mobile robot. An experiment will be performed to measure the effectiveness of each of the reward functions proposed in the previous section. Also, an experiment will be performed to measure the suitability of the proposed reward functions for compound tasks.

4.5.1 Objectives of the experiments

Following are the objectives of the experiments:

- **Experiment 1**: Measure the effectiveness of the reward function proposed in Section 4.3.1 for maintenance tasks.
- **Experiment 2**: Measure the effectiveness of the reward function proposed in Section 4.3.2 for approach tasks.
- **Experiment 3**: Measure the effectiveness of the reward function proposed in Section 4.3.3 for avoidance tasks.
- **Experiment 4**: Measure the effectiveness of the reward function proposed in Section 4.3.4 for achievement tasks.
- **Experiment 5**: Measure the suitability of the proposed reward functions for compound tasks.

4.5.2 Methods and materials

The experiments in this chapter used Webots software to simulate e-puck and create arenas. A reinforcement learning algorithm was written using MATLAB and integrated with Webots to control the mobile robot. The tasks used in the experiments were as generated by Merrick et al. [38], and for consistency, the agent's state and the action space used were also as defined by Merrick et al. [38].

Robot and its Environment

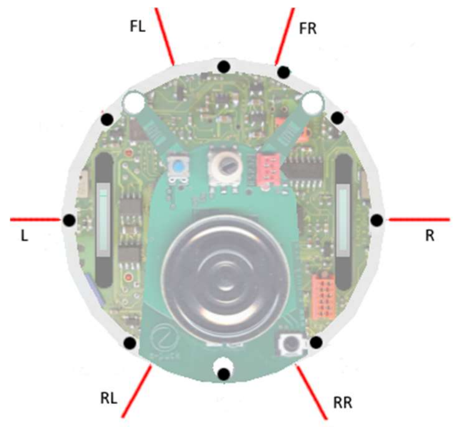


Figure 4.2: Top view of e-puck with labelled proximity sensors. Red lines show the direction in which proximity is detected.

State Vector:

$[\omega^R \ \omega^L \ \theta \ p^L \ p^R \ p^{FL} \ p^{FR} \ p^{RL} \ p^{RR}]$

Actions:

{
1 – left_wheel_speed + δ,
2 – right_wheel_speed + δ,
3 – left_wheel_speed – δ,
4 – right_wheel_speed – δ,
5 – No change to wheel speeds
}

For the experiments in this chapter, the e-puck detailed in Chapter 2 was used. The reinforcement learning state vector used was as defined by Merrick et al. [38]. It consists of six proximity sensors labelled as *Front-Right, Right, Rear-Right, Rear-Left, Left, and Front-Left*. Figure 4.2 shows the top view of the e-puck with the abbreviated labels shown beside the red directional lines along which the sensors detect an obstacle.

The value of the sensor reading indicates the proximity to an object. In the experiments, discretised binary values were used for the proximity sensors. The value 1 indicated that an object was nearby, and 0 indicated no object nearby. The state vector also consisted of angular velocities of the wheels, represented as ω^R and ω^L with the range $-\pi$ to π radians per second. These velocities were discretised into nine values. The robot's orientation angle θ was also used in the state vector, and its range $-\pi$ to π too was discretised into nine values.

Thus, the state of the e-puck comprised nine attributes [ω^R ω^L θ p^L p^R p^{FL} p^{FR} p^{RL} p^{RR}]. The action space comprised five actions: 1) increase the left wheel speed by δ, 2) increase the right wheel speed by δ, 3) decrease the left wheel speed by δ, 4) decrease the right wheel speed by δ, and 5) no change to the left or the right wheel speeds. The value of δ used was $\pi/2$.

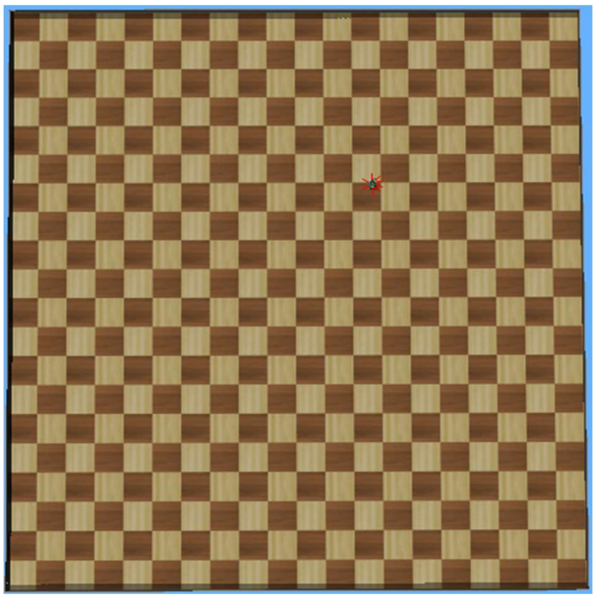


Figure 4.3: Top view of the simple vast walled arena.

The mobile robot environment used for the experiments with primitive tasks was the same as used by Merrick et al. [38]. Figure 4.3 shows the top view of a large 5m × 5m arena that was created for the experiments with the primitive tasks. It is a simple arena with four walls at its periphery and no walls/objects/obstacles in its open area. However, based on the findings from those experiments, two new arenas were created, for the experiment with compound tasks, as detailed in Experiment 5.

Learning Algorithm

In the experiments, a reinforcement learning algorithm called Dyna-Q (detailed in Chapter 2) was implemented using MATLAB. The epsilon-greedy action selection strategy was used for all the experiments, and the epsilon parameter was set to 0.15 with linear decay, i.e. the epsilon was reduced linearly.

Tasks used for the Experiments

For the experiments with the maintenance, approach, avoidance and achievement tasks, the tasks generated by Merrick et al. [38] were used. To generate those tasks, Merrick et al. [38] use Simplified Adaptive Resonance Theory (SART) [110], a clustering technique similar to ART (detailed in Chapter 2), with the main difference being that it works with non-binary values of its data points. Merrick et al. [38] generated two categories of tasks. One category of potential tasks is formed using the experienced states. These tasks can be used as maintenance, approach and avoidance tasks. The second category of tasks is generated using events. These tasks are treated as achievement tasks since the robot aims to achieve those event transitions.

4.5.3 Results and analysis

Experiment 1: Experiment with maintenance tasks

Using the SART clustering technique, Merrick et al. [38] generated two sets of tasks. One set of tasks was based on the clustering of states, and the other set of tasks was based on the clustering of events. For the experiment in this section, tasks based on the clustering of states were used. In this experiment, the tasks are treated as maintenance tasks, i.e. the agent seeks to maintain the task state. The tasks detailed in the 'Task Attributes' column of Table 4.1 are the same maintenance tasks described by Merrick et al. [38]. The 'Task Description' column is merely to provide a human-readable meaning of the task state. Columns M_1 and M_2 are the metrics proposed in this chapter. The column 'Is Task Valid' is an evaluation of whether the task appears to be a valid task or generated in error by the clustering algorithm.

Since the learning phase for a maintenance task is non-episodic, a concept of the trial was used. Each trial lasted for 25,000 steps, and the trial for each task was repeated ten times. Learning was not carried over between trials, i.e. e-puck's state and position were reset. A random starting state was chosen for each trial. Metrics columns show the results averaged over ten trials along with the standard deviation. Equation (4.6) was used as the reward function with the following parameter values: ρ was 0.9, σ was 1, φ was -1, and d was the Euclidian distance.

Table 4.1: Results for maintenance tasks. Metrics M_1, M_2 and reward per episode measured for ten trials with standard deviation shown.

Task Id	Task Attributes	Task Description	Reward per Episode	M_1	M_2	Is Task Valid?
G_1	(2.5, 2.5, 1.8, 0, 0, 0, 0, 0, 0)	Move forward at high speed	493 ± 91	37 ± 8	154 ± 7	Yes
G_2	(0.4, 0.4, 1.2, 0, 0, 0, 0, 0, 0)	Move forward at low speed	568 ± 124	121 ± 25	88 ± 0	Yes
G_3	(-2.4, -2.4, 1.4, 0, 0, 0, 0, 0, 0)	Move backwards at high speed	888 ± 179	88 ± 8	188 ± 9	Yes
G_4	(-0.4, -0.4, -1.3, 0, 0, 0, 0, 0, 0)	Move backward at low speed	866 ± 110	192 ± 28	71 ± 0	Yes
G_5	(0.0, 0.0, -2.8, 0, 1, 0, 0, 0, 0)	Stop for an obstacle in front	3 ±3	1 ± 1	5 ± 0	Yes
G_6	(-0.4, -0.4, 2.9, 0, 0, 0, 0, 0, 0)	Move backward at low speed	601 ± 106	142 ± 24	37 ± 1	Yes
G_7	(-0.8, -0.8, 1.6, 0, 0, 0, 0, 0, 0)	Move backward at moderate speed	848 ± 127	157 ± 26	53 ± 2	Yes
G_8	(0.2, 0.0, 2.4, 1, 0, 0, 0, 0, 1)	Stop for an obstacle behind	0 ± 0	0 ± 0	0 ± 0	Yes
G_9	(0.0, -0.3, 2.1, 1, 0, 0, 0, 1, 0)	Stop for obstacle at left and back	0 ± 0	0 ± 0	2 ± 0	Yes
G_{10}	(-1.9, -1.9, -2.2, 0, 0, 0, 0, 0, 0)	Move backward at moderate speed	763 ± 105	162 ± 23	52 ± 2	Yes
G_{11}	(0.0, 0.0, 3.0, 0, 1, 1, 0, 0, 0)	Stop for an obstacle in front	0 ± 0	0 ± 0	0 ± 0	No
G_{12}	(1.2, 1.2, -2.7, 0, 0, 0, 0, 0, 0)	Move forward at moderate speed	427 ± 85	100 ± 18	36 ± 1	Yes

Once the agent learns to attain the maintenance task, it strives to maintain that task state. There could, however, be reasons why it loses the maintenance attempt. In which case, it will attempt to regain the state; the metric M_1 measures such attempts. A higher value does not necessarily indicate better performance. However, the value of 0 means that the agent did not learn to maintain the task state at all. For example, consider G_1 (move forward at high speed). While in an open area, the e-puck can maintain that state; however, it loses that state when it reaches the wall. The e-puck has to learn to turn around and regain the G_1 state. A higher value of reward per episode metric indicates better agent performance. The metric M_2 counts the longest stretch for which the task state was maintained. A higher value of this metric indicates the continued opportunity that the agent receives in maintaining the task state. For example, tasks G_1 to G_4 require an obstacle-free area. In contrast, tasks G_8 and G_9 require the e-puck to be closer to an obstacle/wall. Thus the design of the arena will determine the opportunity that the agent gets to learn the tasks.

The reward per episode metric is high for G_1 to G_4, G_6, G_7, G_{10} and G_{12}, indicating that the e-puck can maintain those tasks. The corresponding values for metrics M_1 and M_2 are also high for those tasks. Thus, those tasks appear to be easy to maintain. Also, the design of the arena provides the required opportunity to learn to maintain those tasks. For G_{11}, however, all metrics show 0 value. The agent's performance for that task is poor because that task is invalid. For the agent to learn that task, it has to find itself close to a wall in a particular orientation (wall at its *Right* and *Front-Left* but not the *Front-Right*). It is hard to imagine where the e-puck should find itself in the arena to be in such a state. It appears to be unreasonable and marked as such in the 'Is Task Valid?' column. This state could be due to clustering error. It appears that the cluster centroid is not a correct representation of the cluster in this case.

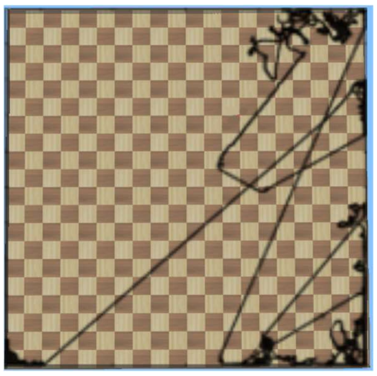


Figure 4.4: Trajectory (shown in black colour) of e-puck learning task G_1. Note the straight-line trajectory. Even during this learning phase, the behaviour of "moving forward at high speed" is apparent.

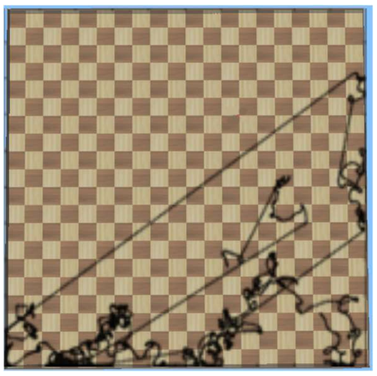


Figure 4.5: Trajectory (shown in black colour) of the e-puck learning task G_3. Note the straight-line trajectory. Even during this learning phase, the behaviour of "moving backwards at high speed" is apparent.

For tasks G_5, G_8 and G_9, which are valid, the metrics M_1 and reward per episode are very low, indicating that the e-puck cannot maintain those tasks. That is not because the maintenance reward design is inadequate, but because of the arena's design. To learn those tasks, the agent requires a wall/obstacle in proximity. Due to the size of the arena, the agent does not get sufficient opportunity to learn and subsequently be able to maintain those task states that require a wall/obstacle in proximity. That is evident from M_2 values, which are low for those tasks. Consider G_5, for example. The agent has to find itself close to the wall at the front. Considering the size of the arena, the chance of such an opportunity is relatively small. That is discussed below by comparing the set of tasks that can be learned in an open area with tasks requiring a wall/obstacle in close proximity.

Figure 4.4 shows the trajectory of the e-puck for task G_1 during one of the learning trials, i.e. the trajectory indicates the learning progress as opposed to the learned behaviour. Admittedly, the trajectory of the e-puck during the learning phase appears uninteresting for most of the tasks; however, as will be seen in Experiment 5 for the compound tasks, the trajectory during the learning phase provides a valuable insight into the learning process. The straight line stretches in Figure 4.4 indicate the maintenance of the high speed while moving forward at a particular orientation.

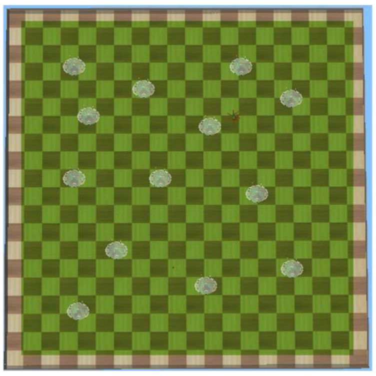


Figure 4.6: The green overlay shows the region where the e-puck could receive a reward for tasks G_1, G_3 and G_{12}. E-pucks are shown scattered to indicate that the location of the e-puck can be anywhere in that region.

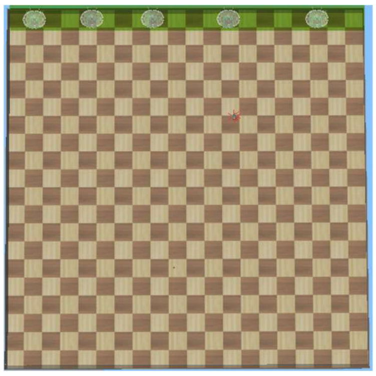


Figure 4.7: The green overlay on the arena shows the region where the e-puck could receive a reward for tasks G_5 and G_8. E-pucks are shown scattered to indicate that the location of the e-puck can be anywhere in that region.

Similarly, Figure 4.5 shows the trajectory of the e-puck learning task G_3. Tasks such as G_1 to G_4, G_6, G_7, G_{10} and G_{12} can only be maintained in the arena's open area. When the e-puck reaches the wall for these tasks, it has to learn to turn around and regain the desired task state. The green overlay in Figure 4.6 shows where the e-puck can maintain these task states. The arena used in the experiment was 5m x 5m in size, and with the sensor range of 0.06m, the chance of the e-puck being in the green zone can be calculated as follows. Consider that the arena is divided into squares of 0.06m. That would create an 83x83 grid. Subtracting one square from all sides (for the region closer to the walls) gives the size of the green overlay as 81×81 grid. The chances of the e-puck finding itself in the green zone is (81×81)/(83×83)=95.2%. That shows that the arena provides the e-puck with plenty of opportunities to learn those task states.

The e-puck does not learn tasks G_5 and G_8. For the e-puck to learn those tasks, it should be near the wall at a particular orientation. The green overlay in Figure 4.7 shows the area where the e-puck should be to learn those tasks. The chances of e-puck being in that region are (81)/(83x83)=1.2%. That shows that the arena does not provide the e-puck with many opportunities to learn those task states. While those tasks are valid, the e-puck does not learn those tasks due to a lack of opportunity. In order to confirm this hypothesis, the experiment for those two tasks was continued in a smaller arena. The arena's size was reduced from 5m x 5m to 0.25m × 0.25m, i.e. a reduction by the factor of 400. Thus, in that smaller arena, the e-puck's chance of finding itself in the right situation is increased 400 times. An experiment in Chapter 6 further explores the usage of specialized arenas, sometimes referred to as a scaffolded environment.

Experiment 2: Experiment with approach tasks

The experiment with approach tasks uses the same set of tasks generated by Merrick et al. [38] as used in the experiment with maintenance tasks. Table 4.2 lists those tasks along with the results for metric M_1 used to measure the effectiveness of the reward design for approach task type. The states detailed under the 'Task Attributes' column are treated as approach tasks. The same as maintenance tasks, learning for approach tasks is non-episodic. Hence, like the previous experiment, the trial constituted 25,000 steps. The trial was repeated ten times for each task, with no learning carried over between them. A random starting state was chosen for each trial. The metric M_1 column shows the results averaged over ten trials and the standard deviation. The reward function, as detailed by Equation (4.7), was used. That equation's parameter values were as follows: ρ was 0.9, σ was 1, φ was -1, and d was the Euclidian distance.

Table 4.2: Results for approach tasks. Metric M_1 measured for ten trials with standard deviation shown.

Task Id	Task Attributes	Task Description	M_1
G_1	(2.5, 2.5, 1.8, 0, 0, 0, 0, 0, 0)	Move forward at high speed	756 ± 16
G_2	(0.4, 0.4, 1.2, 0, 0, 0, 0, 0, 0)	Move forward at low speed	800 ± 21
G_3	(-2.4, -2.4, 1.4, 0, 0, 0, 0, 0, 0)	Move backward at high speed	839 ± 14
G_4	(-0.4, -0.4, -1.3, 0, 0, 0, 0, 0, 0)	Move backwards at low speed	852 ± 11
G_5	(0.0, 0.0, -2.8, 0, 1, 0, 0, 0, 0)	Stop for obstacle in front	884 ± 34

G$_6$	(-0.4, -0.4, 2.9, 0, 0, 0, 0, 0, 0)	Move backward at low speed	874 ± 19
G$_7$	(-0.8, -0.8, 1.6, 0, 0, 0, 0, 0, 0)	Move backward at moderate speed	876 ± 22
G$_8$	(0.2, 0.0, 2.4, 1, 0, 0, 0, 0, 1)	Stop for obstacle behind	873 ± 22
G$_9$	(0.0, -0.3, 2.1, 1, 0, 0, 0, 1, 0)	Stop for obstacle at left and back	860 ± 26
G$_{10}$	(-1.9, -1.9, -2.2, 0, 0, 0, 0, 0, 0)	Move backwards at moderate speed	824 ± 23
G$_{11}$	(0.0, 0.0, 3.0, 0, 1, 1, 0, 0, 0)	Stop for obstacle in front	874 ± 26
G$_{12}$	(1.2, 1.2, -2.7, 0, 0, 0, 0, 0, 0)	Move forward at moderate speed	740 ± 17

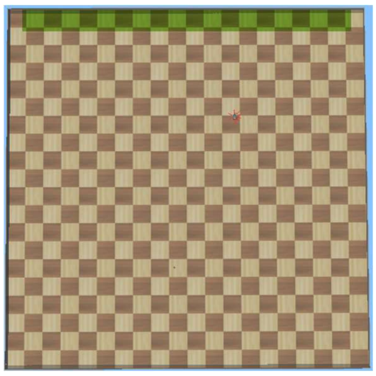


Figure 4.8: The green overlay shows the region where the e-puck could cross the threshold from 'approach' to 'achieve' for tasks G$_5$, G$_8$, and G$_9$.

The metric M_1 shows that the approach tasks are relatively straightforward to attain. Tasks G$_5$, G$_8$ and G$_9$ that were difficult to maintain due to lack of opportunity appear to be easy to attain when treated as approach tasks. That is because the reward function, Equation (4.7), is designed to reward the approach attempt regardless of its distance from the desired task state. Take, for example, G$_5$. That task means stopping for an obstacle in front at a particular orientation. As long as the distance between the current and the desired state reduces, the agent receives a positive reward. The metric M_1 counts the number of instances of that. If the agent reaches too close to the task state, the 'approach' attempt becomes the 'achieve' attempt, resulting in the agent not receiving a positive reward anymore. Tasks G$_5$, G$_8$ and G$_9$ relate to stopping either because the obstacle is to the front or the back. The green overlay in Figure 4.8 shows the region where the 'approach' attempt becomes the 'achieve' attempt for all those tasks.

Experiment 3: Experiment with avoidance tasks

The experiment with avoidance tasks uses the same set of tasks generated by Merrick et al. [38] as used in the experiment with maintenance tasks. Table 4.3 lists those tasks along with the results for metrics M_1 and M_3 used to measure the effectiveness of the reward design for avoidance task type. The states detailed under the 'Task Attributes' column are treated as avoidance tasks. The same as maintenance tasks, learning for avoidance tasks is non-episodic. Hence, like the previous experiment, the trial constituted 25,000 steps. The trial was repeated ten times for each task, with no learning carried over between them. A random starting state was chosen for each trial. Metrics columns show the results averaged over ten trials. Metrics M_1 also shows the standard deviation. The reward function, as detailed by Equation (4.8), was used. The equation parameter values were as follows: ρ was 0.9, σ was 1, φ was -1, and d was the Euclidian distance.

Table 4.3: Results for avoidance tasks. Metrics M_1 and M_3 measured for ten trials with standard deviation shown.

Task Id	Task Attributes	Task Description	M_1	M_3
G_1	(2.5, 2.5, 1.8, 0, 0, 0, 0, 0, 0)	Move forward at high speed	863 ± 14	45
G_2	(0.4, 0.4, 1.2, 0, 0, 0, 0, 0, 0)	Move forward at low speed	805 ± 25	14
G_3	(-2.4, -2.4, 1.4, 0, 0, 0, 0, 0, 0)	Move backward at high speed	753 ± 16	12
G_4	(-0.4, -0.4, -1.3, 0, 0, 0, 0, 0, 0)	Move backward at low speed	762 ± 14	12
G_5	(0.0, 0.0, -2.8, 0, 1, 0, 0, 0, 0)	Stop for obstacle in front	821 ± 30	1
G_6	(-0.4, -0.4, 2.9, 0, 0, 0, 0, 0, 0)	Move backward at low speed	795 ± 25	16
G_7	(-0.8, -0.8, 1.6, 0, 0, 0, 0, 0, 0)	Move backward at moderate speed	775 ± 22	13
G_8	(0.2, 0.0, 2.4, 1, 0, 0, 0, 0, 1)	Stop for obstacle behind	811 ± 18	0
G_9	(0.0, -0.3, 2.1, 1, 0, 0, 0, 1, 0)	Stop for obstacle at left and back	831 ± 17	0
G_{10}	(-1.9, -1.9, -2.2, 0, 0, 0, 0, 0, 0)	Move backward at moderate speed	752 ± 16	6
G_{11}	(0.0, 0.0, 3.0, 0, 1, 1, 0, 0, 0)	Stop for obstacle in front	826 ± 33	0
G_{12}	(1.2, 1.2, -2.7, 0, 0, 0, 0, 0, 0)	Move forward at moderate speed	856 ± 20	7

For avoidance tasks, along with measuring the avoidance attempts M_1, the avoidance performance is also measured by metric M_3, which counts how often the mobile robot failed to avoid the desired task state. The reward function design for the avoidance task type rewards the avoidance attempt, i.e. as long as the agent moves away and stays away from the task state, it is rewarded. As metric M_1 shows, the tasks are relatively easy to attain when treated as avoidance tasks. The reason is that the attempt to avoid the task state is rewarded. The distance from the task state does not matter. As long as it is increasing, the agent is rewarded, i.e. as long as the mobile robot stays away from the task state, the attempt is rewarded positively. For instance, G_3, where the right and left wheel speed attributes values are -2.4. That indicates moving backwards at high speed at a particular orientation indicated by the third state attribute. Thus the task for the agent, as indicated in the 'Task Description' column, is to avoid moving backwards at high speed. When learning to attain that task, the agent receives a positive reward as long as it is able to avoid moving backwards at high speed. That is to say, the agent will receive a positive reward when it is moving forward or even moving backwards but at a low speed. The metric M_1 measures the number of times it receives the positive reward. There will, however, be instances when the agent fails to avoid that state G_3, i.e. during the learning attempt, for one or the other reason, the agent will land in the state that it is supposed to avoid. Those failure instances are measured by metric M_3. The results show that tasks such as G_5, G_8 and G_9 that are difficult to maintain due to lack of opportunity are easier to avoid when treated as avoidance tasks. However, the metric M_3 that counts the failed avoidance attempts is very low for those tasks, indicating that those tasks are difficult in general.

Experiment 4: Experiment with achievement tasks

The second set of tasks generated by Merrick et al. [38] was based on events. Table 4.4 lists those tasks. The 'Task Attributes' column shows the task's event. An event, as detailed in Chapter 2, is a transition represented by $e_t^i = u_t^i - u_{t-1}^i$, where i is one of the state attributes. The experiment in this section uses those tasks as 'achievement' tasks, and to recall, the state vector is represented by the following nine state attributes: [ω^R ω^L θ p^L p^R p^{FL} p^{FR} p^{RL} p^{RR}]. The aim of the achievement task is to achieve the event transition, i.e. the task is considered achieved when the transition e_t^i is reached regardless of what the state attribute u_t^i is. Consider task G_{a5}, for example. As per the description, the task is to increase the right wheel speed by 0.9 and the left wheel speed by 0.6 in a single transition. Thus, the task is considered achieved only when the right and left wheel speed change by that amount in a single transition.

The learning in the case of achievement tasks is episodic. However, for uniformity with the other experiments, the learning attempt for each task was run for 25,000 steps. The metric reward per episode is used to measure the agent's performance. Learning was not carried over between trials, i.e. e-puck's state and position were reset at the start of each trial. A random starting state was chosen for each trial. The reward column shows the results averaged over ten trials and the standard deviation. The column 'Task Description' is a manually provided description of the task to understand what that task means. Equation (4.9) was used as the reward function with the following parameter values: ρ was 0.9, σ was 1, φ was -1, and d was the Euclidian distance.

Table 4.4 shows that five tasks are marked as invalid, and of the remaining, the e-puck was able to learn eight tasks. Firstly, let us analyse the G_{a12}, G_{a18}, G_{a19}, G_{a20} and G_{a22} marked as invalid. For task G_{a12}, task attributes show transition for *Right* and *Front-Left* sensors without any transition for *Front-Right.* It is hard to think of the mobile robot's location in the arena resulting in such a transition. For task G_{a18}, the wheel speed transition of 1.2 and 0.5 radians per second, the change in orientation should be more than |-0.1| radians. It appears that that task is invalid. Regarding G_{a19}, a transition of 2.7 radians per second for the left wheel speed and, regarding G_{a22}, a transition of 2.0 radians per second for the left wheel speed is a big transition for one timestep. The maximum transition in the wheel speeds can only be $\pi/2$ radians. Thus, those tasks are invalid. The results show that nearly a quarter of the tasks are invalid. That raises a question regarding why the task generation technique has nominated such unreasonable transitions as the cluster centroids. The potential reasons could be i) noise, ii) delay in sensing, iii) issue with the simulation of the e-puck resulting in an invalid event or iv) an error in clustering, resulting in cluster centroid not being a correct representation of the cluster. If the issue were with the cluster centroids, the solution would be to place a minimum threshold on the cluster size or shift the cluster centroids to the nearest valid attribute value.

Table 4.4: Results for achievement tasks. Metric reward per episode measured for ten trials with a standard deviation shown.

Task Id	**Task Attributes**	**Task Description**	**Reward per Episode**	**Is Task Valid?**
G_{a1}	(0.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0)	Achieve no change	25000 ± 0	Yes
G_{a2}	(0.0, 0.0, 0.0, 0, 0, 1, 0, 0, 0)	Detect obstacle in front	43 ± 21	Yes
G_{a3}	(-0.1, 0.0, 0.0, 0, 0, -1, 0, 0, 0)	Turn left to avoid an obstacle on the right	0 ± 0	Yes
G_{a4}	(-0.6, 0.0, -0.1, 0, 0, 0, -1, 0, 0)	Turn left to avoid an obstacle on the right	0 ± 0	Yes
G_{a5}	(0.9, 0.6, 0.0, 0, 0, 0, 0, 0, 0)	Increase the speed of both wheels	6521 ± 268	Yes
G_{a6}	(-0.1, 0.1, 0.1, 0, 0, 0, 0, 0, 0)	Turn left	0 ± 0	Yes
G_{a7}	(0.1, 0.0, -0.1, 0, 0, 0, 0, 0, 0)	Turn right	0 ± 0	Yes
G_{a8}	(0.1, -0.4, 0.0, 0, 0, 0, 0, -1, -1)	Turn right to avoid obstacle behind	54 ± 17	Yes

G_{a9}	(-0.3, 0.4, -0.3, 0, 0, -1, -1, 0, 0)	Turn left to avoid an obstacle on the right	0 ± 0	Yes
G_{a10}	(0.0, 0.5, 0.2, 0, 0, 1, 0, 0, 0)	Turn left to detect obstacle on the right	29 ± 16	Yes
G_{a11}	(-0.6, -0.8, -0.2, 0, 0, -1, 0, 0, 0)	Turn right to avoid an obstacle	10 ± 4	Yes
G_{a12}	(0.0, 0.7, 0.3, 0, -1, 1, 0, 0, 0)	Turn left to sense obstacle on the right	0 ± 0	No
G_{a13}	(0.2, -0.8, -0.4, 0, 0, 0, 0, 1, 0)	Turn right to sense obstacle on left	12 ±4	Yes
G_{a14}	(0.0, 0.6, 0.1, 0, 0, 0, 0, 1, 1)	Turn to detect obstacle behind	0 ±0	Yes
G_{a15}	(0.0, -0.1, 0.0, 0, 1, 1, 0, 0, 0)	Turn right to sense an obstacle in front	0 ±0	Yes
G_{a16}	(1.0, 0.5, 0.1, 0, 1, 0, 0, 0, 0)	Turn right to sense obstacle on left	0 ±0	Yes
G_{a17}	(0.7, 0.9, 0.3, 0.0, -1, 0, 0, 0, 0)	Turn left to sense obstacle on left	18 ±3	Yes
G_{a18}	(1.2, 0.5, -0.1, 0, -1, 0, 0, 0, 0)	Turn to avoid an obstacle on left	0 ±0	No
G_{a19}	(0.2, 2.7, -0.2, 0, -1, 0, 0, 0, 0)	Turn to avoid an obstacle on left	0 ±0	No
G_{a20}	(-1.7, -0.5, 0.1, 0, 1, 0, 0, 0, 0)	Turn to detect obstacle on the right	0 ±0	No
G_{a21}	(-0.7, -1.2, -0.3, 0, 1, 0, 0, 0, 0)	Turn to detect obstacle on left	0 ±0	Yes
G_{a22}	(1.4, 2.0, 0.2, 0, 0, 0, 0, 0, 0)	Turn left	0 ±0	No

The rest of the tasks are valid. However, results show that only eight of the seventeen show non-zero reward per episode. Tasks G_{a2}, G_{a8}, G_{a10}, G_{a11}, G_{a13}, and G_{a17} could be achieved only a few times, whereas the tasks G_{a4}, G_{a9}, G_{a14}, G_{a16}, and G_{a21} could not be achieved at all. That is not because the reward function is insufficient but due to the lack of learning opportunities. For example, to achieve G_{a2} (where the task is to stay still when an object is detected in front) or G_{a8} (where the task is to turn right when an object is detected behind the robot), the e-puck has to find itself close to the wall in a particular orientation to achieve those transitions. Similarly, to achieve G_{a4} (where the task is to learn to turn left to avoid an obstacle on the right) or G_{a16} (where the task is to turn right to sense the object on the left), again, the e-puck has to find itself near the wall in a particular orientation to achieve those transitions. Considering the size of the arena and the fact that walls are the only objects in the arena, the chances of e-puck finding itself in those situations is very slim, which results in the lack of learning opportunities. In the discussion for the maintenance tasks (Experiment 1), the solution to such an issue was to reduce the arena's size. That solution will work for these tasks as well.

Tasks G_{a3}, G_{a6}, G_{a7}, and G_{a15} are valid but could not be achieved. The reason for this appears to be the choice of the granularity of discretisation used in this experiment. Consider G_{a3}, for example. The task attribute for right wheel speed is a transition of -0.1. Considering that the range of $-\pi$ to π for wheel speeds and orientation was discretised into nine values. That difference of -0.1 (between s_{t-1} and s_t) is too fine-grained to be detected by the relatively coarser nine values state space design and, as a result, will be treated as 0.0. The same is the case for G_{a7}. The task is to turn right by increasing the right wheel speed by 0.1. Discretizing the range of -π to π (i.e. 2π radians) into nine buckets gives the granularity of 0.7 radians. That makes any change less than 0.7 radians difficult to detect. One of the solutions to this problem would be to discretise the wheel speed and orientation attributes into more than nine values. That would increase the size of the state space and the corresponding size of the Q-table and thus increase the time it will take for the mobile robot to learn those tasks, which is not a problem per se; however, it was not done to keep the state space uniform with other experiments in this chapter.

While experiments 1 to 4 validate that the proposed reward functions can be used for primitive tasks, they have also highlighted flaws in the task generation technique, the choice of state attributes and action space and granularity of discretisation. Based on the observations, the following changes were made to the setup for the experiments in the next subsection: i) the size of the arena was reduced, ii) a task-specific environment was created, i.e. walls and obstacles were introduced to provide more learning opportunities, iii) the orientation attribute was removed from the reinforcement learning state vector resulting in more intuitive achievement type primitive tasks, for example, ‘achieve turning left’, and iv) the action space was simplified.

Experiment 5: Experiment with a combination of tasks

Not all tasks are atomic. Consider the task of booking a meeting as detailed in [111]. As such, it would appear that it can be represented as an achievement task. However, consider a variant of such a task, i.e. booking a recurring meeting. Since people’s schedules change or the meeting has to be adjusted for some participants due to the daylight savings time, keeping the meeting current in the calendar of all the participants becomes a follow-on task. Thus, the actual task is a sequence of tasks of different types and is better modelled as “achieve then maintain”. Consider a mobile robot example where the robot’s task is to follow a wall. As such, this example appears to be an atomic task. However, when one considers the different start positions of the robot and the contour of the wall, it becomes apparent that it is better to model the task as a sequence of tasks. The first task is to approach a wall; the second task is to maintain a set distance from the wall. Other tasks would be negotiating a corner, a cul-de-sac, or going around the wall opening. Such a task is a compound task and best represented as "approach then maintain" in the simplest case. The corresponding reward function can be built from the reward functions of the constituent primitive tasks.

A compound task can be a sequential or a concurrent combination of the constituent primitive tasks. Further, as will be seen in Chapter 6, a third way is implemented using modular reinforcement learning. It is not related to how primitive tasks are combined but when primitive skills are triggered. The experiment in this section, which is akin to modular reinforcement learning, demonstrates the suitability of the proposed primitive reward functions to learn compound tasks. It demonstrates the learning of a compound task by using handcrafted if-then-else rules to identify and trigger different primitive reward functions proposed in this chapter. It is, however, interesting to think about how this can be done autonomously. That is discussed as an avenue for future work in Chapter 7. The experiment also demonstrates how such a combination of tasks is a sample-efficient learning method compared to learning the complex skill from scratch. That is further explored in Chapter 6.

State Vector:

$[\omega^R \ \omega^L \ p^L \ p^R \ p^{FL} \ p^{FR} \ p^{RL} \ p^{RR}]$

Actions:

{
1 – Turn Left,
2 – Step Forward,
3 – Turn Right
}

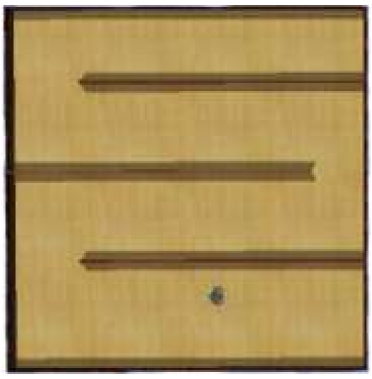


Figure 4.9: Top view of the maze arena.

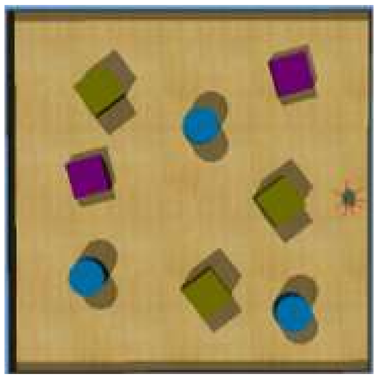


Figure 4.10: Top view of the arena with obstacles.

For this experiment, two new 2m x 2m arenas, as shown in Figure 4.9 and Figure 4.10, were created. As shown in Figure 4.9, the maze arena is an arena with walls to form a simple maze. The e-puck learns the compound task of following a wall in this arena. The learning is directed using Function 1, which details the if-then rules for the task.

Function 1) Wall following task in the maze arena

```
if wall on the left
    achieve turning left
elseif wall close on the left
    maintain moving forward
elseif wall on the right
    achieve turning right
elseif wall close on the right
    maintain moving forward
elseif wall at the front and left  /* i.e. corner on the left */
    achieve turning right
elseif wall at the front and right  /* i.e. corner on the right */
    achieve turning left
elseif wall at the front
    achieve turning right
elseif no wall nearby
    maintain moving forward
end
```

Figure 4.10 shows the arena with cylindrical and cuboid-shaped obstacles. In this arena, the e-puck learns the compound tasks of learning to avoid obstacles using Function 2, which details the if-then rules to attain that task. In this experiment, the state and action space was modified based on the observations from the previous experiments. The state space does not include orientation, and the action space comprises three actions: turn left, step forward and turn right.

Function 2) Obstacle avoidance task in the arena with obstacles

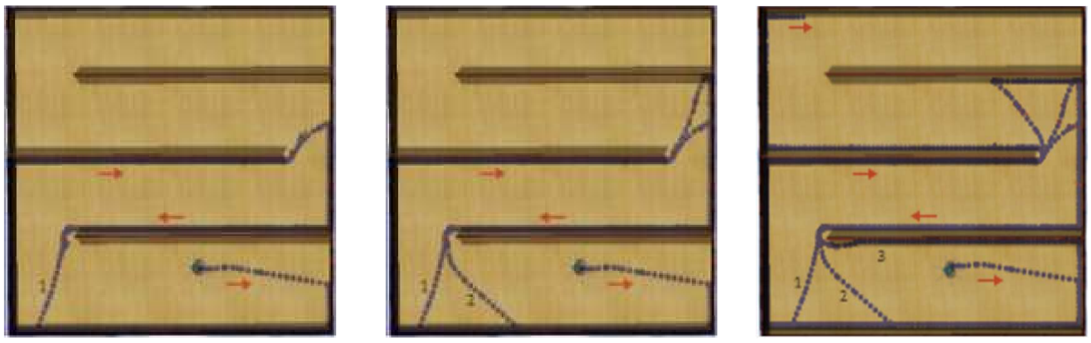
```
if obstacle on the left
    achieve turning right
elseif obstacle on the right
    achieve turning left
elseif obstacle at the front and/or side
    achieve turning right
elseif obstacle at the back
    maintain moving forward
elseif no obstacle anywhere nearby
    maintain moving forward
end
```

Table 4.5 shows the results for the two tasks detailed above. Metrics M_1, M_2 and reward per episode are used to measure the agent's performance for the compound tasks. The table shows the average metric value (with a standard deviation) for each task run ten times, with a trial length of 25,000 steps. In the experiment, these metrics measure the cumulative reward gained by the e-puck for all the primitive tasks combined, i.e. they measure the reward for the compound task.

The metric 'reward per episode' shows the overall learning progress and indicates that the agent has learned both tasks. Also, the significant value of metric M_1 (number of times the task maintenance attempt is regained) indicates that the e-puck learns to regain the task state. Metrics M_2 (the longest stretch of task maintenance), which depends on the design of the arena, shows that the e-puck is able to maintain the task state for a significant number of time steps.

Table 4.5: Results for compound tasks. Metrics M_1, M_2 and reward per episode measured for ten trials with standard deviation shown.

Task Description	**Reward per Episode**	M_1	M_2
Wall following	16833 ±115	1373 ±29	78 ±6
Avoiding obstacles	13613 ±109	747 ±24	81 ±8



(a) (b) (c)

Figure 4.11: (a), (b) and (c) show different stages of the e-puck learning the wall-following task in the maze arena. The blue line is the trajectory of the e-puck, with the red arrows showing the e-puck's direction. The e-puck starts at the bottom third of the arena, goes straight until it is close to a wall and then follows the wall to its left. Attempts 1, 2, 3 are the trajectory of the e-puck trying to go all the way around the wall. Similarly, at the top half of the arena, it takes the e-puck four attempts to go all the way around the wall and then it continues following the wall.

Figure 4.11 shows the trajectory for one of the trials to follow the wall in the maze arena while the agent is in the learning phase, i.e. it is not the trajectory of the learned behaviour. Function 1 comprises if-then-else rules to trigger a specific skill to reach the wall, follow the wall to the left, negotiate corners and go all the way around the wall. It is a hand-crafted combination of achievement and maintenance task types, each of which is triggered in a specific situation. The red arrow shows the direction of the e-puck. Initially, when there is no wall in close proximity, it goes straight until it reaches a wall. That then triggers a new skill of following the wall to the left. The attempt to go all the way around the wall seems to require several attempts. That is indicated by the digits 1, 2, and 3 besides the trajectory. For a reinforcement learning agent, that is an order of magnitudes quicker than the agent attempting to learn the skill of following a wall from scratch. That validates the suitability of proposed reward functions for learning compound tasks, i.e. the primitive reward functions can be combined to form a skill for the compound task. Also, it indicates that the proposed metrics are suitable to measure the agent's performance for those compound tasks.

4.6 Summary

Reward in reinforcement learning guides the learning. Those rewards, in most cases, are handcrafted and often task-dependent. However, for dynamic environments, the tasks to learn are not known in advance. Thus, for open-ended and continuous learning, the agent architecture should be equipped with the ability to guide the learning with task-independent reward. This chapter proposed reward functions for the 'achievement', 'approach', 'avoidance' and 'maintenance' tasks, a categorisation of tasks based on the functional aspect of how they are considered 'attained'. The reward design exploits the inherent property of the type of the task, and hence the reward is task-independent.

This chapter also proposed metrics to measure the agent's performance for these task types. Those metrics clearly indicate that the e-puck is learning to attain the desired agent state in the experiments. Experiments also showed that such reward design could be used to 'attain' compound tasks where the constituent primitive tasks use the proposed reward functions.

When such a task-independent reward design is integrated with a task generation mechanism capable of generating compound tasks, it will result in an autonomous agent capable of learning primitive as well as compound tasks. The literature review shows a lack of a suitable autonomous compound task generation mechanism. The next chapter will investigate the possibility of equipping the agent to generate tasks of varying complexity.

CHAPTER 5 SELF GENERATION OF TASKS TO DIRECT THE LEARNING

Parts of this chapter have been published in: P. Dhakan, K. Kasmarik, I. Rano, and N. Siddique, "Open-Ended Continuous Learning of Compound Goals," IEEE Transactions on Cognitive and Developmental Systems, vol. 13, no. 2. pp. 274–285, 2019.

5.1 Introduction

Chapter 3 introduced an agent architecture for open-ended and continuous learning. Continuous learning is an essential aspect for the agent to learn multiple skills over its lifetime, but it is the open-ended learning that directs that learning. By that, it means whether the agent increases the knowledge of its environment by gaining the breadth of the knowledge first and then the depth or learns by using some hybrid approach depends on the tasks generated by the task generation module. That is to say, to create an open-ended learning agent, the meta-cognitive aspects such as 'what to learn' [19] [20] and 'when to learn' [20] need to be considered. This chapter focuses on the question of 'what to learn'. Figure 5.1 is the Modular Continuous Learning Architecture proposed in Chapter 3. The 'Task Generation Module', shown in green colour, is the focus of this chapter.

Several task generation approaches have been proposed to provide an agent with the capability to decide 'what to learn'. Broadly they appear to be techniques that either generate subtasks of the provided high-level task or generate just a flat high-level task. In the case of subtask generation, the over-arching task is supplied, and the aim is to find its subtasks. The literature review shows that this can be done by determining the frequency of visited states [66], based on the identification of bottleneck states and graph partitions that enables the transition in the state space [112] [113], based on relative novelty when attempting to attain an overarching task [114] or by sequencing the skills [115]. The flat high-level task generation techniques are where the agent looks for the changes to its environment in real-time that can then be considered potential tasks [69] [63] [116] [53] [117], and offline techniques where the tasks are generated based on the novel states experienced during the agent's exploration of its environment [38]. Chapter 2 showed the different categorisation of tasks, one of which was based on the complexity of the task ranging from 'primitive' to 'compound' tasks. From the literature review on self-task generation, it is apparent that the existing task generation techniques do not focus on the composition aspect of the task. That is to say, they do not focus on how a compound task is built using its constituent primitive tasks. This chapter's contribution is a task generation technique that fulfils that gap.

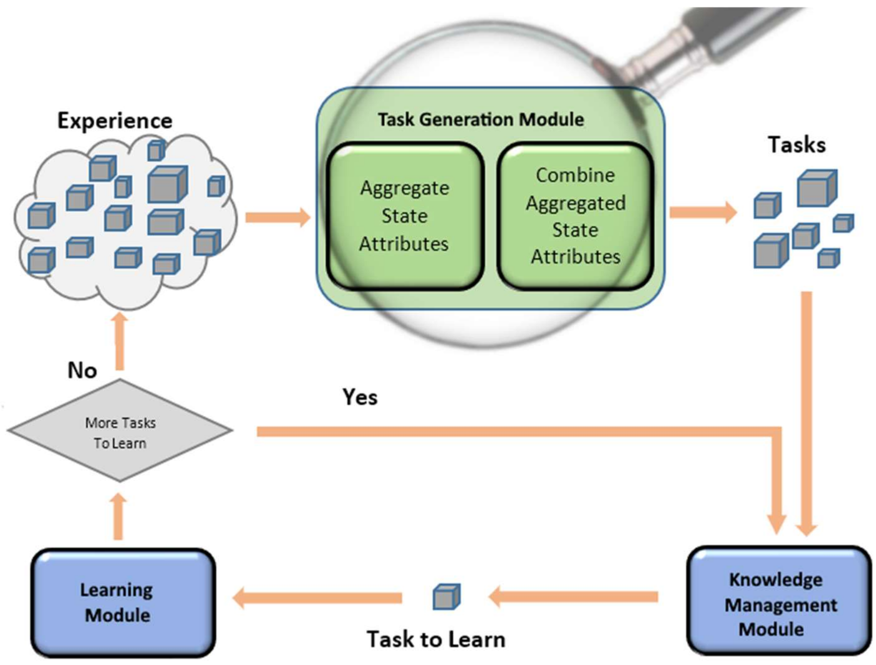


Figure 5.1: Modular Continuous Learning Architecture revisited. Task generation, the focus of this chapter, is a contribution related to the Task Generation Module.

This chapter proposes a domain-independent technique to generate tasks of varying complexity. The tasks are generated by combining the low-level units that make up the agent's state. Using a granularity parameter, the size of these units can be varied from coarser to finer units. These units are then used to create tasks of varying complexity. Using simulated e-puck based experiments, this chapter will show how the agglomerative hierarchical clustering is used to aggregate the state attributes, which are then used to form tasks. It will also show how these primitive tasks can then be combined to form compound tasks. The rest of this chapter is organised as follows: Section 5.2 reviews the literature on task generation. Section 5.3 will detail the proposed task generation mechanism. Section 5.4 will detail the methodology and the results of the experiments, and finally, Section 5.5 will provide the concluding remarks.

5.2 Self-Generation of Tasks

As seen in Chapter 3, to progress reinforcement learning based agent architecture beyond single-task learning, the architecture should be combined with a mechanism that can generate a stream of tasks for the system to learn, i.e. make the architecture open-ended. That is what directs the learning of the system and provides it with 'what to learn'. This section will review the literature on the self-generation of tasks.

'Task', as seen in Chapter 2, is defined as an objective that an agent should attain. Also, as seen, they are categorised based on whether they are: i) hard or soft, ii) state-based or action-based, and iii) low-level or high-level. The organisation of this literature review is based on the third category. It details the techniques that generate the top-level tasks and techniques that generate sub-tasks given a top-level task.

5.2.1 Task generation

Intrinsic motivation reviewed in the previous section can be used to explore the agent's state space. Task generation using novelty [118] or curiosity [119] [53] is one approach. Forestier et al. [116] propose an intrinsically motivated task exploration technique. In this, the agent iteratively samples the continuous and high-dimensional state space and sets a task. A novelty heuristics then generates an intrinsic reward that is used to learn the task. Further, this is used to ascertain the learnability of the task leading to the tuning of the task exploration preferences. Baranes and Oudeyer [54] propose a task generation mechanism using adaptive curiosity or a measure of interest. Marsland et al. [69] proposed an online novelty detection algorithm that aims to discover novel situations from the states it has experienced. A neural network is trained to ignore the normal perceptions that have been experienced before. This leaves anything that has not been sensed before being treated as 'novelty'. These novel situations are treated as tasks that the agent should aim to attain.

Santucci et al. propose a multi-layer GRAIL agent architecture [12] seen in the previous section. Task generation is one of the layers of this architecture that detect a change in its environment. These changes, if unique, are considered tasks. Mirolli and Baldassarre [89] argue that skill acquisition should be cumulative and best represented using a hierarchical structure for complex skills. In that case, one substructure, using knowledge-based intrinsic motivation, can determine what to learn, and another substructure, using competence-based motivation, can decide which task to learn.

Other approaches include the one shown by Rolf et al. [120], where the system auto-generates tasks using inconsistencies during exploration. Hanheide et al. [63] have shown how a service robot uses the states it has explored before to determine the gaps in the knowledge, which are then used as tasks. Merrick et al. [38] proposed experience-based task generation, where a mobile robot explores its environment to gather experience data. Using a simplified adaptive resonance theory (SART), the data points, i.e. experienced states, are clustered. The cluster centroids are then treated as tasks.

5.2.2 Subtask generation

As an alternative to task generation reviewed in the previous subsection, the subtask or interim task or provisional task generation techniques are intended to generate a hierarchical task structure with an assumption that the end task is provided. The subtask generation, also referred to as subtask discovery, is not so much seen as a means for an agent to learn multiple tasks but as a means to divide and conquer the monolithic end task. When used with reinforcement learning, the reward for the subtasks act as interim rewards, simplifying the reward design and accelerating learning. Santucci et al. [121] suggested that instead of using intrinsic motivation directly for skill acquisition, it should be used to generate an interim concept of tasks, and those tasks can then direct the acquisition of the skill. Simsek and Barto [112] hypothesise that certain states are central in navigating the environment for any problem. They term such measure of structural centrality as “betweenness”. The states with a higher measure of betweenness than their neighbouring states are identified as potential subtasks. Such subtasks help to navigate the interaction graph of the environment. Thus, when the skills to achieve the subtasks are combined, they enable reaching other regions of the interaction graph and eventually the end task.

In reinforcement learning, commonly, an option [42] is used to represent a macro action. The end state of the option can be considered as a subtask. The discovery of options has been an area of active research [122] [67]. Konidaris and Barto [115] extended the usage of options to the continuous domain and proposed “skill chaining”, a subtask discovery mechanism for reinforcement learning agents. Like Simsek and Barto [112], they too hypothesise that a useful skill always lies in the solution path of the end task, i.e. a skill for a useful subtask is critical to solving the main task. In the proposed technique, the reinforcement learning agent works backwards. Starting with the end task, the agent creates a short-range option to reach that end state and learns the skill to reach that end state. The initial state of that option is then considered the next end task, and the aim of the agent is to discover another short-range option and the skill to reach that state. This continues until, eventually, the agent reaches the initial starting state, thus identifying all the subtasks along the path. The learned skills are then chained, which forms the solution to traverse from the starting state to the end state.

Recently, the usage of tasks to direct learning has also drawn the interest of the deep learning community. Andrychowicz et al. [75] proposed training a deep neural network on automatically generated interim tasks using the concept of experience replay. They showed that the reinforcement learning agent could learn end tasks for the cases when the rewards are sparse and even when those end tasks have never been observed during the training. In a learning framework proposed by Held et al. [123], they automatically generated interim tasks at a difficulty level that is just appropriate for the agent. That curriculum of tasks directs the agent’s learning and enables the agent to learn a variety of skills without any previous knowledge of its environment.

5.2.3 Gap in the state-of-the-art

While most of the task generation and the sub-task generation techniques found in the literature can be integrated into the architecture proposed in this thesis, the review of the literature showed that the current research either focused on generating top-level tasks or focused on discovering subtasks given a top-level task. That highlights a research opportunity as summarized in Table 5.1.

Table 5.1: Table showing the focus areas of each category reviewed in this section.

Focus area	**Task generation techniques**	**Subtask generation techniques**
Generate flat/high-level tasks	✓	✗
Generate subtasks of a given high-level task	✗	✓
Generate tasks of varying complexity	✗	✗

The existing techniques do not focus on generating tasks that vary in complexity. For a continuous learning agent, it is essential that the agent is constantly learning new skills. For that, it should be able to self-generate tasks of varying complexity ranging from simpler, more primitive tasks to compound tasks that are, in essence, some combination of primitive tasks. Such a task generation would enable the development of a learning curriculum for the autonomous agent. That raises the question, how does one design a module to self-generate tasks of varying complexity? The following section will aim to answer that question.

5.3 Self-Generation of Tasks of Varying Complexity

Providing the system with the ability to decide 'what to learn' is essential for it to be called an open-ended learning system. While any task generation mechanism can be plugged into the Modular Continuous Learning Architecture, the literature review shows that no mechanism exists that enables the system to generate tasks of varying complexity. Tasks can be called 'primitive' if they are atomic in nature and cannot be further subdivided and 'compound' if they are composed of two or more primitive tasks. This section proposes a task generation mechanism that can generate tasks of varying complexity ranging from primitive to compound tasks.

5.3.1 Step 1: Gather experience

The task generation process starts with gathering experience data. In this stage, the agent explores its environment by randomly moving around. The observations are recorded in terms of states experienced by the agent. The agent's location and the salient features of the environment may or may not be part of the agent's state vector. Hence the main aim of this random exploration is not that the agent should visit all parts of its environment; instead, for the agent to visit as many states of its internal state space as possible, akin to body babbling as termed in developmental robotics literature. These visited states are recorded as an experience and form data points for the next step of the process. The state of the agent can be represented as a vector of its state attributes:

$$s = [u^1, u^2, \ldots, u^n] \tag{5.1}$$

where u^i_t is typically a numerical value that describes the agent's internal or external attribute, and n is the number of attributes in the state vector. The experience will contain duplicate data points since the same state will be experienced several times. During the next step, which is to cluster these data points, only unique data points are used. This is so that the dominant states do not overpower the seldom visited states.

5.3.2 Step 2: State attribute aggregation using hierarchical clustering

The next step is to aggregate the related state attributes to create logical units or groups which can be seen as regions in the state space. For this, hierarchical agglomerative clustering [124], a bottom-up clustering algorithm, is used. The algorithm starts by treating each data point as a cluster. Consider cluster f_p and a data point u^x. The cluster can be represented as:

$$f_p = \{ u^x \} \tag{5.2}$$

The algorithm then carries out bottom-up clustering by successively merging the pairs of clusters. This iterative process continues until all clusters have been combined into one big cluster consisting of all the data points. This merging occurs based on the linkage criteria that specify dissimilarity between the clusters. The algorithm can use the following linkage criteria: 'complete, 'average', 'weighted', 'centroid', 'single' and 'ward'. In the proposed technique, the 'complete' criterion is used. The 'complete' criterion, also referred to as the farthest neighbour criterion, uses the largest distance between the data points in the two clusters. The resulting cluster can be represented as:

$$f_{p+q} = max\{\, dist(u^x, u^y) : \ u^x \in f_p, \ u^y \in f_q \,\} \tag{5.3}$$

where f_{p+q} is the cluster formed using the 'complete' linkage between the two clusters, f_p and f_q, and $dist$ is the distance function of the pairwise distances of the data points in the cluster. The commonly used distance functions are Euclidean, Hamming, and Manhattan, to name a few. Alternatively, a custom distance measure can also be used. The iterative clustering process is then carried out until all the data points are combined into a single cluster.

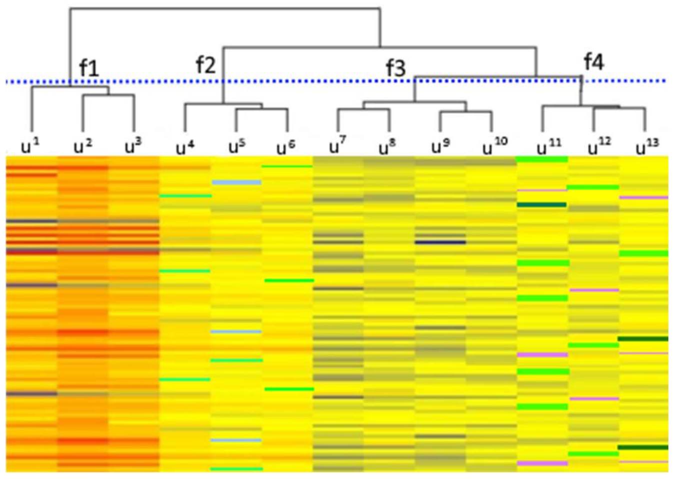


Figure 5.2: A sample dendrogram is shown at the top of the figure. The rest of the figure (shown in colour) shows the state attribute data points. The aggregations of state attributes are shown by clusters f_1, f_2, f_3 and f_4.

The result of hierarchical agglomerative clustering is represented in a hierarchical tree of clusters called a dendrogram. The dendrogram shows an edge between the two closest clusters merged at a particular hierarchy level. As shown in Figure 5.2, the data points are arranged such that the state attributes $u_t^1, u_t^2, u_t^3, \ldots, u_t^n$ (i.e. the attributes that make up a state vector) are arranged as columns, and the values of the state attributes which make up the experienced state appear as rows. Initially, each state attribute $u_t^1, u_t^2, u_t^3, \ldots, u_t^n$ is considered a cluster of its own. As per hierarchical agglomerative clustering, iteratively, the pairs of clusters are merged until all clusters have been combined into one big cluster. That means the algorithm's output is hierarchical clusters of state attributes, which can be visualised as regions within the state space, as shown in Figure 5.3.

The height of the edge of the dendrogram corresponds to the distance between the two clusters. The algorithm takes the threshold criterion as an input which can be specified in terms of the height of the dendrogram's edge or the total number of desired clusters. In the proposed technique, the total number of clusters is specified. Figure 5.2 shows the threshold criterion using a blue dotted line, resulting in four clusters. Depending on the threshold criterion, either fewer but coarser clusters or many fine-grained clusters are generated. The clustered/grouped state attributes, i.e. regions in the state space, can be represented as $\{f_1, f_2, \dots f_i\}$ where f is the cluster, i.e. aggregation of two or more state attributes, i is the number of clusters identified due to the aggregation of state attributes such that $i < n$ and n is the number of state attributes.

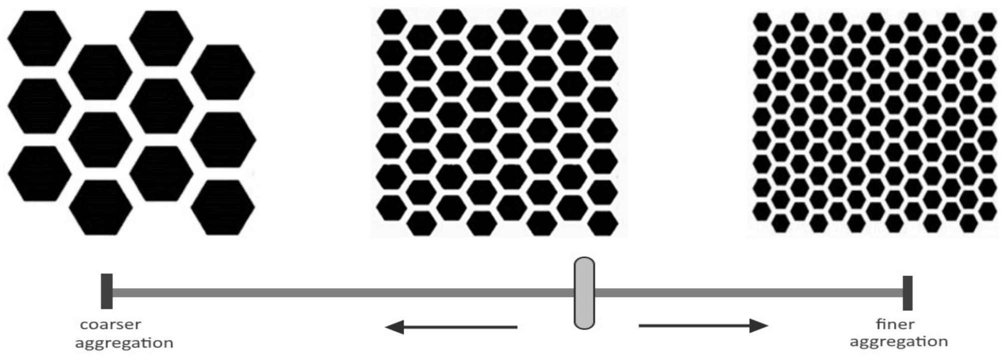


Figure 5.3: A symbolic representation of the state attributes aggregation resulting in regions within the state space. The figure shows state-space in a 2D representation, and hexagons represent the regions within the state space. Varying the threshold criteria results in coarser to finer aggregations/groups.

The aggregation reduces a larger set of state attributes to a smaller set while preserving most of the information represented by the large set, hence reducing cardinality [124]. Such compressed representation of the state can be shown as:

$$s = [f_1, f_2, \dots, f_i] \tag{5.4}$$

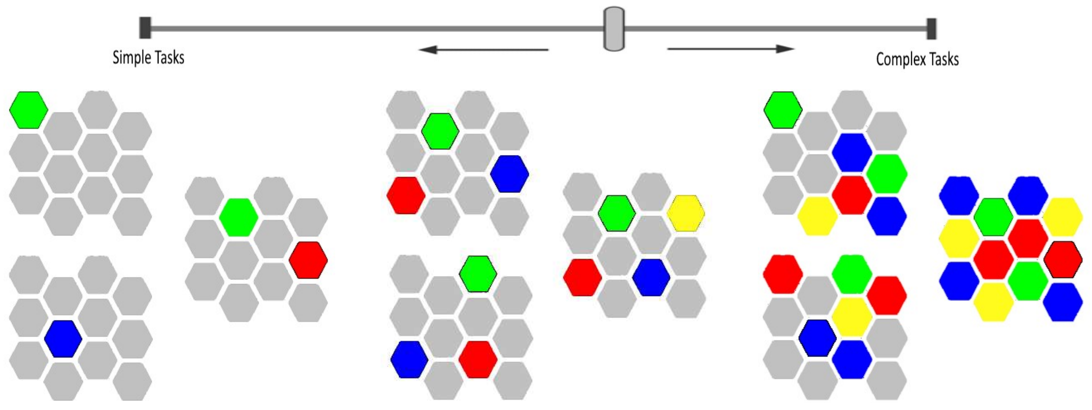
These aggregations are then further processed by the next step of the proposed technique to generate tasks of varying complexity.

5.3.3 Step 3: Generate tasks

The next step in the process is to generate tasks. This step is further divided into the following:

i. enabling the region(s) of the state space by providing it with a value. These values can be a maximum, minimum or average value that the respective attributes can take, or if one has the domain knowledge, they can be the domain-specific value.

ii. combining the above mentioned enabled regions using operators such as ‘OR’ and ‘AND’ to form a valid state vector.



$max(f_1)$,

$avg(f_2)$,

$min(f_3) \wedge max(f_4)$,

…

$max(f_1) \wedge avg(f_2) \wedge min(f_3)$,

$min(f_2) \wedge avg(f_3) \wedge max(f_4)$,

$avg(f_3) \wedge max(f_4) \wedge min(f_5) \wedge avg(f_6)$,

…

$max(f_1) \wedge min(f_2) \wedge avg(f_3) \wedge max(f_4) \wedge min(f_5)$,

$avg(f_3) \wedge max(f_4) \wedge min(f_5) \wedge min(f_6) \wedge max(f_7)$,

$min(f_4) \wedge max(f_5) \wedge avg(f_6) \wedge max(f_7) \wedge avg(f_8) \wedge max(f_9)$,

…

Figure 5.4: Tasks, ranging from simple to complex, generated by enabling and combining different groups/aggregations. A few sample tasks are shown in the figure. The green, blue, yellow and red colour indicates that the aggregation is enabled using one of the functions such as *min*, *max* and *avg*. The grey colour indicates that the aggregation is not enabled and will be masked/ignored.

The resulting state vector is one of the states in the state space of the agent. As detailed in Chapter 4, such a state can then be treated as a state to maintain, approach, or avoid resulting in the state being called a maintenance, approach, or avoidance task. When just the single region is enabled, it results in a relatively simple task, and when multiple regions are enabled and combined, it creates a more complex task, as shown in Figure 5.4.

Table 5.2 lists the sample tasks. The first few rows list primitive tasks, and the following rows list complex tasks. To generate these tasks, it is assumed that the state space is a discretised metric state space. The functions 'minimum', 'maximum', 'average', and 'specific value' are used to enable the aggregated group. The groups are then combined by applying operators such as 'OR' and 'AND'. Any combination of the operators can be used, and any number of groups can be chained together, resulting in a procedural generation of rich and expressive tasks.

Table 5.2: Examples of the tasks that can be generated using the proposed task generation mechanism.

Task	Task Description
$\boldsymbol{min}(\boldsymbol{f_1})$	Minimize the value of the group
$\boldsymbol{max}(\boldsymbol{f_1})$	Maximize the value of the group
$\boldsymbol{avg}(\boldsymbol{f_1})$	Attain an average value of the group
$\boldsymbol{specific}(\boldsymbol{f_1}, \boldsymbol{value})$	Attain a specific value of the group
$\boldsymbol{min}(\boldsymbol{f_1}) \wedge \boldsymbol{min}(\boldsymbol{f_2})$	Minimize group1 as well as group2
$\boldsymbol{max}(\boldsymbol{f_1}) \vee \boldsymbol{max}(\boldsymbol{f_2})$	Maximize either group1 or group2 or both
$\boldsymbol{max}(\boldsymbol{f_1}) \vee \boldsymbol{max}(\boldsymbol{f_2}) \wedge \boldsymbol{min}(\boldsymbol{f_3})$	Maximize group1 or maximize group2 and minimize group3
$\boldsymbol{max}(\boldsymbol{f_1}) \wedge \boldsymbol{max}(\boldsymbol{f_2}) \wedge \mathbf{max}(\boldsymbol{f_3}) \wedge \boldsymbol{max}(\boldsymbol{f_4}) \wedge \boldsymbol{max}(\boldsymbol{f_5}) \wedge \boldsymbol{max}(\boldsymbol{f_6})$	Maximize group1 and maximize group2 and maximize group3 and maximize group4 and maximize group5 and maximize group6

5.3.4 Step 4: Task pruning

The task generation process starts with states experienced by the agent. Those experience data points are then clustered, processed and procedurally combined to generate tasks. This processing results in tasks that are the states that the agent may never have experienced before. That may mean that the generated tasks can be invalid or difficult to reach states. Also, as shown in Table 5.2, such task generation results in a combinatoric explosion resulting in a vast number of tasks. That would hinder instead of guiding the learning of the overall system detailed in Chapter 3. This issue is not unique to the technique proposed in this chapter but a problem of the techniques that use procedurally-generated tasks, as acknowledged in [125] and [126]. Wang et al. [125] and the open-ended learning team [126] use procedurally generated environments for their open-ended learning agents to increase their general competence. However, a sizeable percentage of the environments are invalid or unlearnable and have to be discarded. Considering that, the next step for the proposed technique is to prune the tasks. For this, selection criteria should be used that prunes the potential tasks resulting in a final list of tasks. The criteria can be domain-dependent, requiring domain knowledge or some external intervention. Alternatively, the criteria can be intrinsic motivation based to keep the task generation process autonomous. For example, the criteria can be based on: i) novelty, in which case, only the potential tasks with high dissimilarity with other tasks are selected as tasks, or ii) current competency level of the agent's skill for similar tasks, in which case, only the similar potential tasks are selected as tasks. The rest of the potential tasks are categorised as non-tasks. Figure 5.5 shows the detailed steps of the task generation technique.

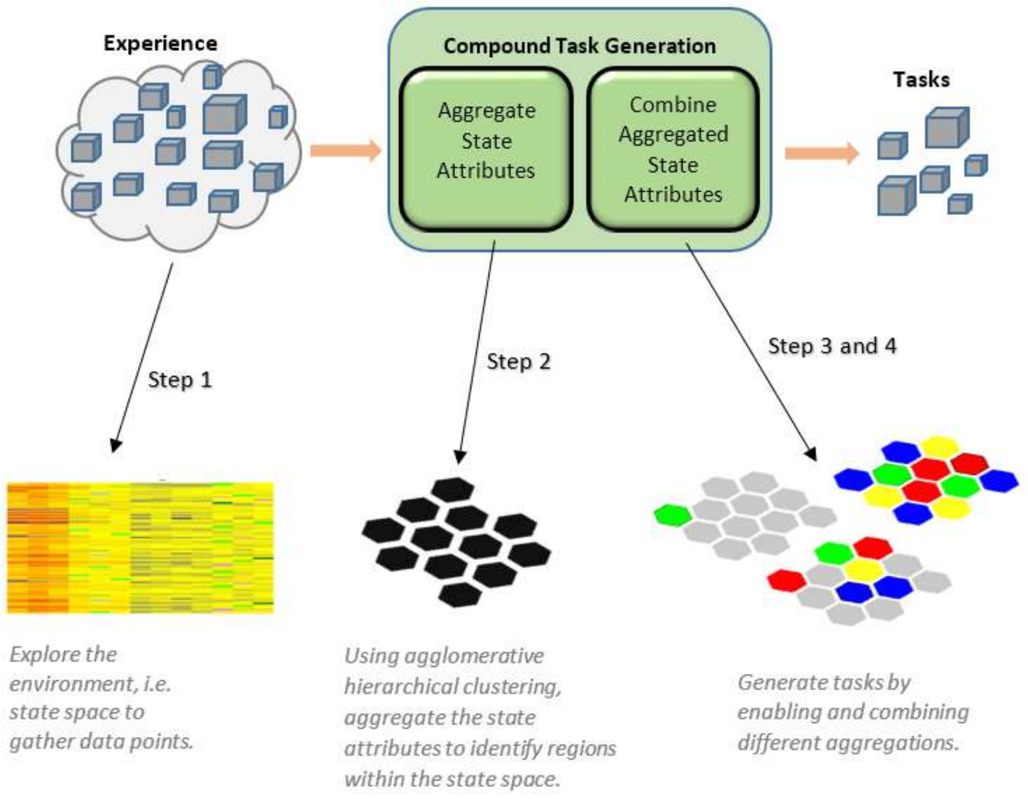

Experience
Compound Task Generation
Aggregate State Attributes
Combine Aggregated State Attributes
Tasks
Step 1
Step 2
Step 3 and 4
Explore the environment, i.e. state space to gather data points.
Using agglomerative hierarchical clustering, aggregate the state attributes to identify regions within the state space.
Generate tasks by enabling and combining different aggregations.

Figure 5.5: The detailed steps of the proposed task generation technique.

5.3.5 Integration with continuous learning architecture

When the threshold (i.e. the cut-off) criterion of the hierarchical clustering algorithm is varied, it results in aggregations of different granularity, as shown in Figure 5.3. Low cut-off would result in fewer but coarser aggregations, and high cut-off would result in several fine-grained aggregations. When those aggregations are used to generate tasks, it may result in different tasks compared to the tasks generated with the previous level of aggregation. It, however, does not render the previously generated tasks obsolete. All the aggregations are added to the list of unique aggregations, and the aggregations of different sizes can be combined as long as it forms a valid state vector. Thus the proposed technique is suitable for continuous learning. The proposed technique can be integrated with the agent architecture detailed in Chapter 3. Algorithm 5.1 shows this integration.

Algorithm 5.1: Task Generation and Learning Cycle

```
Start
  do

    /* Experience Gathering */
    for steps = 1: max_exploration_steps
      Gather experience by interacting with the environment
    end for

    /* Task Generation */
    /* Aggregate State Attributes */
    if state attribute aggregation is not already done
      Aggregate state attributes
    end if
    /* Create Tasks */
    Combine aggregated state attributes to form tasks
    Store the tasks in the task_list

    /* Task Learning */
    for task = 1: task_list
      Learn the task
      Store the learned knowledge
    end for

  while (environment has changed)

end
```

5.3.6 Examples of tasks of varying complexity

To discuss the examples of tasks of varying complexity, consider an autonomous vacuum cleaner, a practical application of a mobile robot. The reinforcement learning state of such a machine would comprise values of the following: proximity sensors, accelerometer, vacuum-cleaning motor and battery charge indicator. A simple task for the vacuum cleaner can be to clean the floor irrespective of the inclination of the surface, i.e. on a ramp or a level surface. In this case, the task representation ignores the accelerometer values. A more complex task can be cleaning only the level floor, i.e. avoid any surfaces with a considerable inclination. In this case, the task representation considers the accelerometer and proximity sensor values. Another simple task could be to clean the floor while avoiding obstacles irrespective of the battery charge level. In this case, the vacuum cleaner will not attempt to make its way back to the charging station when the battery level dips below a certain level. A more complex task can be to consider the battery charge level. Such a task would be represented as cleaning the floor while avoiding obstacles, avoiding inclined surfaces, and being aware of the battery charge level. These use-cases can be extended to other mobile robot applications such as a lawnmower.

As another example, consider a humanoid robot whose reinforcement learning state vector comprises the speed and orientation of the motors in its arms and legs, the orientation of the motor in its neck, accelerometer and some form of representation of what its camera sees. A relatively simple task, albeit not primitive, can be that the robot walks forward on a flat surface at a constant gait. A more complex task can be that it walks forward on a flat surface while carrying an object in its arms. Another complex task can be climbing a ramp/steps or walking on an uneven surface while carrying/balancing an object in its arms.

5.4 Mobile Robot Experiments

The previous section proposed a task generation technique. The experiments in this section will validate that technique. The experiments will use the e-puck mobile robot. The robot will randomly move around in its environment to gather experience. Those experience data points will then be used to generate tasks of varying complexity.

5.4.1 Objectives of the experiments

The objectives of the experiments in this section are:

- Validate the task generation.
- Update the environment dynamically to check if the newly created unique aggregated state attributes can be integrated with the previous list, keeping the previously generated tasks valid.

5.4.2 Methods and materials

The experiments in this chapter used an e-puck mobile robot. To simulate the e-puck and to create arenas, Webots was used. The reinforcement learning agent was written using MATLAB and integrated with Webots to control the mobile robot.

Robot and its Environment

For the experiments in this chapter, the e-puck's eight proximity sensors labelled: *Front-Right, Right-Diagonal, Right, Rear-Right, Rear-Left, Left, Left-Diagonal, Front-Left*, and three ground sensors labelled *Left, Centre, Right* were used. Discrete binary values were used for both proximity and ground sensors. Also, the wheel speeds represented as ω^L and ω^R were used in the state vector to represent the speeds of the left and the right wheels, and they too were discretised to binary values. The state vector for the experiments was represented as $[\omega^L\ \omega^R\ p^{FR}\ p^{RD}\ p^R\ p^{RR}\ p^{RL}\ p^L\ p^{LD}\ p^{FL}\ g^L\ g^C\ g^R]$. The action space for the experimentation comprised 'Turn Left', 'Step Forward' and 'Turn Right'. Figure 5.6 shows a sketched top view with the labelled proximity sensors, ground sensors and wheels.

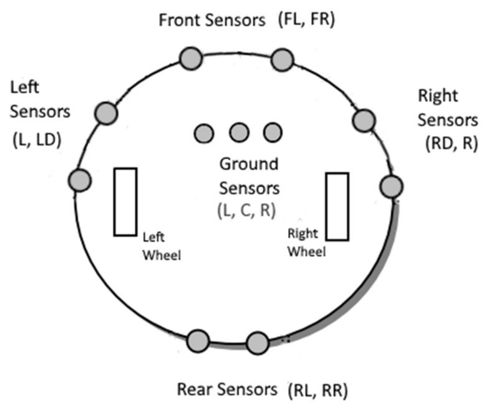


Figure 5.6: A plan view representation of e-puck with all its state attributes (proximity sensors, ground sensors and wheels) labelled.

State Vector:

$[\omega^L\ \ \omega^R\ \ p^{FR}\ \ p^{RD}\ \ p^R\ \ p^{RR}\ \ p^{RL}\ \ p^L\ \ p^{LD}\ \ p^{FL}\ \ g^L\ \ g^C\ \ g^R]$

Actions:

{
1 – Turn Left,
2 – Step Forward,
3 – Turn Right,
}

For the experiments, the same three arenas as detailed in Chapter 3 were used. Figure 5.7, Figure 5.8 and Figure 5.9 show the maze arena, circular arena with tracks, and arena with obstacles.

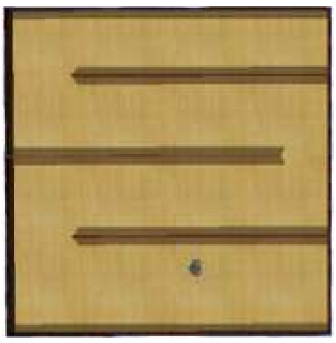


Figure 5.7: Top view of the maze arena.

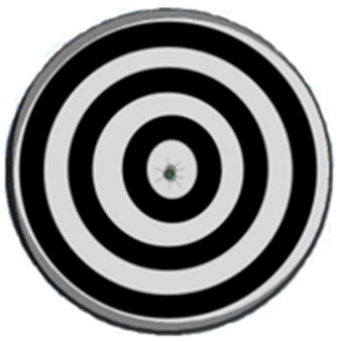


Figure 5.8: Top view of the circular arena with tracks.

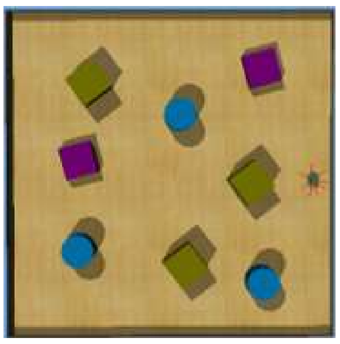


Figure 5.9: Top view of the arena with obstacles.

Learning Algorithm

For the experiments, a reinforcement learning algorithm called Dyna-Q (detailed in Chapter 2) was implemented using MATLAB. The epsilon-greedy action selection strategy was used for the experiments, and the epsilon parameter was set to 0.1 with linear decay.

5.4.3 Results and analysis

Step 1: Gather experience

The first step for the e-puck mobile robot was to gather the experience data points. For this, the robot was made to wander around in its environment. The aim here is not for the robot to explore as much of the arena as possible but to explore its internal state space as possible. This step is akin to body babbling. In this phase, the robot essentially follows a reinforcement learning exploration policy where it does not receive a reward for any action. For this, the epsilon-greedy action selection strategy's epsilon parameter is set to 1, which encourages exploration of its state space. When the robot collides with an object/wall, Webots simulates the dynamics of the environment resulting in the next state of the robot. This phase was continued for 10,000 steps, and the data points were recorded.

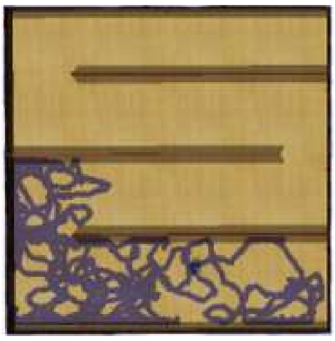


Figure 5.10: Trajectory, shown in blue colour, of the e-puck randomly exploring the maze arena. The states experienced during this exploration would be related to "being close to a wall", "being in an open space", to name a few.

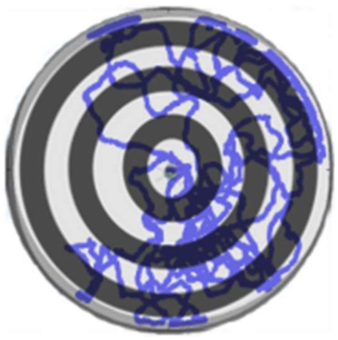


Figure 5.11: Trajectory, shown in blue colour, of the e-puck randomly exploring the circular arena with tracks. The states experienced during this exploration would be related to "being on a track", "not on a track", to name a few.

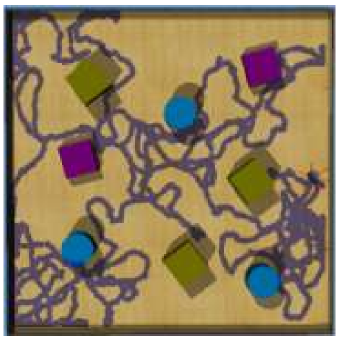


Figure 5.12: Trajectory, shown in blue colour, of the e-puck randomly exploring the arena with obstacles. The states experienced during this exploration would be related to "being close to an obstacle", "being in an open space", to name a few.

A similar exploration was carried out for all three arenas, as shown in Figure 5.10, Figure 5.11 and Figure 5.12. Of these 10,000 data points, a list of unique data points was generated. That is so that the frequently observed states and the states experienced only a few times have an equal representation. This list of unique data points formed the input for the next step. The size of this list depends on how many unique states of its internal state space the e-puck was able to visit during the experience gathering phase and can vary between 1 and 10,000.

Step 2: State attribute aggregation

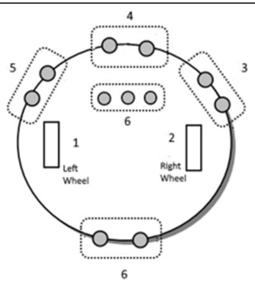
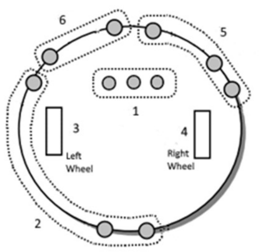
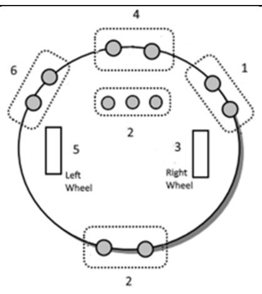
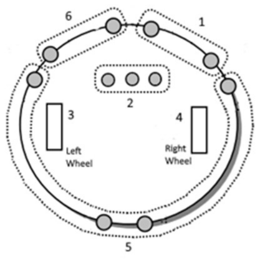
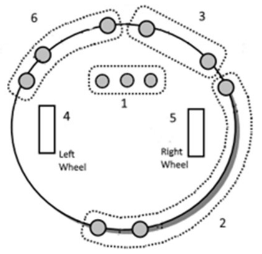
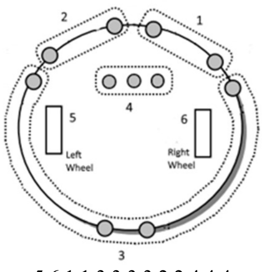
The next step was to cluster the state attributes for which the proposed technique uses hierarchical agglomerative clustering. The clustering algorithm requires a distance metric to calculate the dissimilarity between the data points and decide which cluster the data point should be added to. In the experiments, Euclidean distance was used as a distance measure. The clustering algorithm also requires a linkage criterion, essentially a metric that calculates the dissimilarity between the cluster pairs. That distance measure is then used to link up the clusters in a bottom-up manner. In the experiments, 'complete' linkage was used, which represents the largest distance between the two clusters. The clustering technique can be run i) in a batch mode, i.e. cluster all the data points, ii) in online mode, i.e. cluster a constant stream of data points, or iii) in a semi-online mode where a small set of data points are clustered in one round. The clustering was run in the semi-online mode in the experiments. That was done to demonstrate that the proposed technique is suitable for continuous learning. The algorithm also requires a cut-off/threshold criterion, which is used to determine the number of clusters that should be generated. In the experiments, that number was set to 6.

Table 5.3 shows the output of the agglomerative clustering. The five rows for each of the arenas are the output of the five rounds of clustering (semi-online mode). Each time the algorithm generates six clusters, i.e. assigns the state attributes to one of the clusters. Consider the first row for the maze arena for which the output is "1 2 4 3 3 6 6 5 5 4 6 6 6". That output means that:

- the first state attribute (ω^L, i.e. the left wheel speed) is added to cluster #1,
- the second state attribute (ω^R, i.e. the right wheel speed) is added to cluster #2,
- the fourth and fifth state attributes (*Right-Diagonal* and *Right* proximity sensors) are added to cluster #3,
- the third and the tenth state attributes (*Front-Right* and *Front-Left* proximity sensors) are added to cluster #4,

- the eighth and ninth state attributes (*Left* and *Left-Diagonal* proximity sensors) are added to cluster #5, and
- the last three state attributes (ground sensor attributes) are added to cluster #6.

Table 5.3: Output of hierarchical agglomerative clustering for the maze arena, the circular arena and the arena with obstacles. Row #1 and #3 also contain a graphical view of the clustering technique's output in terms of aggregation of the e-puck's state attributes.

Iteration #	Maze Arena	Circular Arena	Arena with Obstacles
			
1	1 2 4 3 3 6 6 5 5 4 6 6 6	3 4 5 5 5 2 2 2 6 6 1 1 1	5 3 4 1 1 2 2 6 6 4 2 2 2
2	3 1 2 4 4 6 6 5 5 2 6 6 6	3 6 2 2 5 5 5 4 4 1 5 5 5	5 3 4 4 6 6 6 2 2 1 6 6 6
			
3	3 4 1 1 5 5 5 5 6 6 2 2 2	4 5 3 3 2 2 2 6 6 6 1 1 1	5 6 1 1 3 3 3 3 2 2 4 4 4
4	3 4 1 2 2 5 5 6 6 6 5 5 5	4 5 6 6 3 3 3 2 2 2 1 1 1	5 6 2 2 4 4 4 4 1 1 3 3 3
5	3 4 5 5 2 1 1 1 6 6 2 2 2	4 5 6 6 3 3 3 2 2 2 1 1 1	5 6 4 4 2 2 2 2 3 3 1 1 1

The first and the third row of the table also show a graphical view of the aggregation of the state attributes, i.e. proximity sensors, ground sensors and wheel speeds. The digit near the group is the cluster number. As seen in the graphical representation, some groups consist of just a single state attribute, whereas others consist of more than two state attributes. The results show that the output of the clustering algorithm may be the same regardless of the arena. To understand this consider the maze arena and arena with obstacles. Both arenas have walls; hence the experienced data points may be similar, leading to the same output by the clustering algorithm.

The proposed technique then takes this output from the clustering algorithm and identifies unique groups/aggregations. For example, the ground sensors always seem to belong to the same cluster, forming a unique aggregation. From the output shown in Table 5.3, all the unique groups of the state attributes are extracted. Table 5.4 shows those unique groups. The column 'Aggregated State Attributes' shows the attribute(s) in the group. The description column is a manually added description to provide an intuition as to what that group means and is not required for the working of the technique. The state attribute in the group is represented with a '#', and all other attributes are represented using a '-'. As an example, consider just the following two outputs 1 2 4 3 3 6 6 5 5 4 6 6 6 and 3 1 2 4 4 6 6 5 5 2 6 6 6. The unique aggregations from that are # - - - - - - - - - - - -, - # - - - - - - - - - - -, - - # - - - - - - - # - - - and so on.

Table 5.4: Unique groups of state attributes obtained from further processing the results shown in Table 5.2.

Group Id	Aggregated State Attributes	Description
f1	# – – – – – – – – – – – –	Group consisting of just the left wheel attribute
f2	– # – – – – – – – – – – –	Group consisting of just the right wheel attribute
f3	– – # # # – – – – – – – –	Group consisting of front right and the two right sensors
f4	– – # # – – – – – – – – –	Group consisting of front right and the right diagonal sensor
f5	– – # – – – – – – # – – –	Group consisting of the two front sensors
f6	– – # – – – – – – – – – –	Group consisting of the front right sensor
f7	– – – # # – – – – – – – –	Group consisting of the two right sensors
f8	– – – – # # # # – – – – –	Group consisting of the right, two back and left sensors
f9	– – – – # # # – – – # # #	Group consisting of the right, two back and three ground sensors
f10	– – – – # # # – – – – – –	Group consisting of right and two back sensors
f11	– – – – # – – – – – # # #	Group consisting of right and the three ground sensors
f12	– – – – – # # # – – – – –	Group consisting of two back and the left sensor
f13	– – – – – # # – – – # # #	Group consisting of two back and three ground sensors
f14	– – – – – – – # # # – – –	Group consisting of two left and front left sensors
f15	– – – – – – – # # – – – –	Group consisting of two left sensors
f16	– – – – – – – – # # – – –	Group consisting of left and left front sensors
f17	– – – – – – – – – # – – –	Group consisting of just the front left sensor
f18	– – – – – – – – – – # # #	Group consisting of the three ground sensors

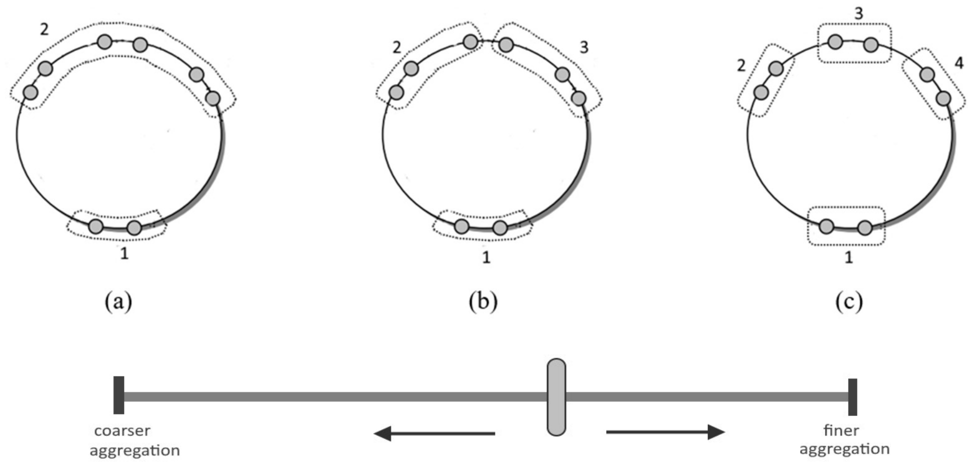


Figure 5.13: Varying levels of aggregation of the e-puck's proximity sensors.

During this step, varying the cluster threshold criterion will result in a varying granularity of aggregation. Figure 5.13 shows a graphical representation of how varying the threshold criterion results in the varying levels of aggregations of the e-puck's proximity sensors.

Step 3: Generate tasks

The final step of the proposed technique was to generate tasks. That was done by generating a state vector with one or more groups shown in Table 5.4 enabled. A group is considered enabled when it has a value assigned to it. For example, all the attributes in the group are assigned a maximum value. If the group is not enabled, it is ignored. The other way to enable a group is to: (i) assign a minimum value to all the attributes in the group and (ii) assign a set value to all the attributes in the group. Assigning a set value may require domain knowledge. Only '*min*' and '*max*' functions were used in this experiment. Further, as per the proposed technique, these groups are to be combined using operators to form a valid state vector. Any number of groups can be chained together as long as they form a valid state vector. This experiment used 'AND' and 'OR' operators.

Table 5.5 shows sample tasks. They vary in complexity from simple, more primitive tasks to complex. The 'Task' column details which group was used to create those tasks and how the groups were combined, i.e. *min* or *max* functions are applied to the relevant group using '#'. Value of '0' was used when the function *min* was applied, and '1' was used when the function *max* was applied. That was because all the state attributes (wheel speeds, proximity and ground sensors) in this experiment use discrete binary values. The task description is also provided in the table to indicate what that task means. A manual semi-structured approach was used to create this task list. In this approach, the aim was not to generate all possible tasks (as this leads to combinatorial explosion) but to ensure that all the aggregated state attributes f_1 to f_{18} are used. The tasks were generated using both 'AND' and 'OR' operators, and each of the aggregated state attributes f_1 to f_{18} was used at least once. The aim was to ensure that there is a good representation of both primitive and compound tasks formed by combining multiple aggregated state attributes. Based on that, seventeen tasks G_1 to G_{17} were generated. For example, G_{11} (- - - - - - 0 0 1 1 1 0 0 0), which was created by minimizing f_{13} (- - - - - # # - - - # # #), i.e. (- - - - - 0 0 - - - 0 0 0) 'AND' maximizing f_{14} (- - - - - - - # # # - - -), i.e. (- - - - - - - 1 1 1 - - -).

All the tasks in this table were considered maintenance tasks for which the reinforcement learning agent learned the skills. The columns 'Metric – M_1' (the same metric proposed in Chapter 4) and 'Reward per Episode' are the metrics to measure agent performance. Metric M_1 measures how often the maintenance attempt was regained, and reward per episode measures the total positive reward gained by the agent during the 25,000 step learning trial. The trial for each task was run ten times. Metrics columns show the results averaged for those ten trials, and the standard deviation was recorded as shown in Table 5.5.

Table 5.5: A list of handcrafted compound tasks (created in a semi-structured way) and results from learning those tasks. Metrics M_1 and reward per episode measured for ten trials with standard deviation shown.

Task Id	Task	Description of the task	Metric – M_1	Reward per Episode	Is Task Valid?
G_1	$min(f_1)$	Turn left or stay still	1466 ± 44	23258 ± 60	Yes
G_2	$max(f_3)$	High sensor values of *Front-Right* and *Right* sensors	305 ± 135	17423 ± 2783	Yes
G_3	$max(f_5)$	High sensor values of front sensors	45 ± 18	24364 ± 587	Yes
G_4	$min(f_8)$	Low sensor value of the *Right*, back, and *Left* sensors	65 ± 34	23950 ± 1205	Yes

G5	$max(f_{10})$	High sensor values of *Right* and back sensors	0 ± 1	0 ± 3	Yes
G6	$min(f_{14})$	Low sensor values of *Left* and *Front-Left* sensors	89 ± 35	20227 ± 1586	Yes
G7	$min(f_1) \land min(f_2)$	Stay still	0 ± 0	0 ± 0	No
G8	$min(f_3) \lor min(f_4)$	No obstacle/wall at the front and right	69 ± 45	21533 ± 1483	Yes
G9	$min(f_7) \land max(f_8)$	No obstacle/wall at the right, but obstacle /wall at the back and left	0 ± 0	0 ± 0	Yes
G10	$min(f_9) \lor max(f_{10})$	Obstacle/wall at the right and back while the robot is not on a track	0 ± 0	0 ± 0	Yes
G11	$min(f_{13}) \land max(f_{14})$	No obstacle/wall on the back but obstacle/wall on the left and front left while the robot is not on a track	338 ± 174	21265 ± 2252	Yes
G12	$max(f_{15}) \lor min(f_{16})$	Obstacle/wall on the left	378 ± 134	7939 ± 1893	Yes
G13	$max(f_1) \land max(f_3)$	Move forward or turn right, obstacle/wall at the front and right	1441 ± 182	18488 ± 2799	Yes
G14	$max(f_5) \lor max(f_7)$	Obstacle/wall at the front and right	645 ± 238	15980 ± 3317	Yes
G15	$min(f_2) \land min(f_4) \land min(f_6)$	Stay still or turn right; no obstacle/wall on the front and right	1172 ± 235	17243 ± 5801	Yes
G16	$max(f_1) \lor max(f_2) \land max(f_{18})$	Move forward while on a track	113 ± 81	23121 ± 1469	Yes
G17	$max(f_1) \land min(f_5) \lor max(f_{14}) \land min(f_{17})$	Move forward or turn right, obstacle/wall on the left and no obstacle/wall at the front	720 ± 179	4889 ± 1346	Yes

The metrics show that the e-puck learns most but not all tasks. As mentioned above, the tasks were generated in a semi-structured way. Domain knowledge was not used in generating those tasks; hence not all generated tasks may be valid. That validity is indicated by the "Is Task Valid?" column. Also, a task, which is the robot's state, may not have been visited during the exploration phase and could be a state that is difficult to reach. Results show that the e-puck is not able to learn G_7, G_9 and G_{10}. The task G_7 means that irrespective of the proximity sensor values or the ground sensor values, it is to stay still, i.e. regardless of its position on the board, whether it is near a wall or an obstacle, whether it is on a black region on the ground or not, it is to stay still. As per the design of the action space in this experiment, there is no way the e-puck can stay still. The only valid actions are to turn left, step forward or turn right. Thus, task G_7 is not a valid task. Task G_9 means that the e-puck is supposed to maintain a state where there is no obstacle/wall to the right, but there is a wall close to its back. This task is valid however appears difficult to learn.

Similarly, task G_{10} means that the e-puck is supposed to maintain a wall/obstacle on its right side and behind while not on a black region on the ground. This task is valid; however, it also appears difficult to learn. If the proposed technique is used to implement the task generation module of the agent architecture detailed in Chapter 3, then the knowledge management module of the architecture can be responsible for pruning the tasks that are invalid or very difficult to learn given the design of its action space. The metrics show that the rest of the tasks are learnable.

The task description column provides an intuition of the meaning of the tasks that are generated by the proposed technique. Rather than a semi-structured approach to generate the tasks, a structured approach could be used if one had the domain knowledge, resulting in the generation of useful tasks. Regardless, varying the cut-off criterion would result in coarser to finer aggregation units. That and the way these aggregations are combined would result in tasks of varying complexity. That validates the proposed task generation technique. Also, as shown in the above experiment, the whole process can be run iteratively without making the previous aggregations obsolete. Thus, this technique is suitable for continuous learning.

5.5 Summary

To exhibit open-ended learning, the agent should be able to self-generate tasks. As the system learns solutions to 'attain' those tasks, its overall knowledge increases. However, not all tasks in the real world are primitive or flat. Hence, the task generation mechanism should be able to generate tasks of varying complexity—primitive as well as compound. This chapter proposed such a mechanism. The process is divided into the following phases: i) explore the environment and store the experience as data points, ii) using aggregation technique, discover salient groups in the data points, and iii) using domain-independent knowledge, form general tasks or using domain knowledge, form tasks specifically valuable for that domain. In this chapter, the state attribute aggregation is done using hierarchical clustering, whose parameters can be varied to create fewer but coarser clusters or a larger number of fine-grained clusters. When the task generation step is carried out using these aggregations, it results in tasks of varying complexity—thus filling the gap in the literature.

This chapter also showed that the whole process could be repeated in a cycle when the environment changes. The resulting aggregated state attributes can be assimilated with previous unique aggregations; thus, newer aggregations do not make the previously generated tasks obsolete. This is important not only to preserve the system's integrity but also to continuously improve its knowledge of the environment without having to restart all over again when something changes in the environment. Also, it is essential, especially for the robotics domain, where it is not always possible to learn all the skills from scratch due to the sample inefficiency of reinforcement learning. The agent architecture proposed in Chapter 3 is continuously learning, i.e. constantly increasing the overall knowledge. It would be worthwhile to leverage the learned knowledge and reuse it to know the solutions to future tasks. That could be an alternative approach to reinforcement learning’s sample inefficiency and would result in a continuous learning system more suitable for the real world. That is the subject of the next chapter. It will investigate how the learned knowledge can be reused to learn solutions for future tasks.

CHAPTER 6 REUSE OF LEARNED KNOWLEDGE BY SKILL COMPOSITION

Parts of this chapter have been published in: P. Dhakan, K. Kasmarik, P. Vance, I. Rano, and N. Siddique, "Concurrent Skill Composition using Ensemble of Primitive Skills." IEEE Transactions on Cognitive and Developmental Systems, Accepted for publication – 10th May 2022.

6.1 Introduction

A key characteristic of an open-ended learning system is that it can learn to perform multiple tasks. For that, it should be able to determine what tasks it should learn [19] [20] and also be able to exploit the learned knowledge to improve the performance of the task at hand [127]. Chapter 5 proposed a task generation technique so the agent can select 'what to learn', which was the contribution related to the task generation module of the architecture proposed in Chapter 3. Once the agent has built up a repository of skills, a logical next step is to enable it with the capability to combine the skills it has already learned to solve new tasks. In reinforcement learning, which is used in the learning module of the architecture, an agent learns by interacting with its environment. In many cases, the amount of interaction needed to learn a task is quite large [29], making it impractical for many robotics applications. While the reinforcement learning community is conscious of this drawback and has proposed sample-efficient algorithms [41] and, different approaches such as imitation learning [33] [76], transfer learning [128] [129] are used. However, further benefits can be gained by reusing the previously learned knowledge to create a solution for future tasks. The literature review shows that these approaches have not previously been considered in open-ended learning architectures, specifically in an intrinsically motivated reinforcement learning setting. That is the contribution of this chapter.

This chapter will detail how the previously learned skills can be combined to form skills for new tasks. The responsibility of deciding what skills to combine and how to combine those skills lies with the architecture's knowledge management module. This chapter's contribution is thus related to that module. Figure 6.1 shows the knowledge management module in green, highlighting the focus of this chapter.

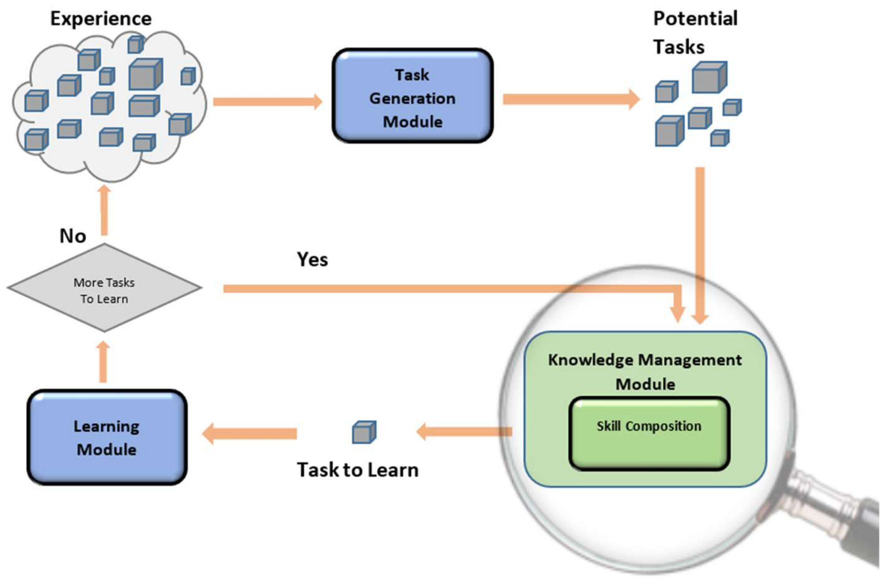


Figure 6.1: Modular Continuous Learning Architecture revisited. Skill composition, the focus of this chapter, is a contribution related to the Knowledge Management Module.

Chapter 4 showed that one of the task categorisations is whether they can be considered atomic. A task is ‘primitive’ if it cannot be further divided into simpler tasks and ‘compound’ if it is formed by a combination of simpler tasks. That combination can be ‘sequential’, i.e. multiple primitive tasks are carried out in a sequence, or ‘concurrent’, i.e. multiple primitive tasks are carried out at the same time (i.e. interleaved in parallel). The literature review showed that option is a common method used to combine the tasks in sequential order in reinforcement learning. A primitive skill can be packaged as an option, and many such options can be sequenced together to form a skill for a compound task. The literature review also showed a few ways of combining the skills concurrently, namely using the Gaussian Mixture Model [130] and a Mixture of Experts [131] [132]. Integration of the concept of options into the architecture proposed in Chapter 3 gives it the capability to sequentially combine the skills for the primitive tasks without the need for any additional task-dependent knowledge. However, integrating the concurrent combination techniques found in the literature review requires additional task-dependent knowledge or human intervention. Inspired by Modular Neural Networks [133], this chapter proposes a method of combining reinforcement learning policies represented using a neural network. It is a method to combine the primitive tasks concurrently without requiring other task-dependent knowledge. When such skill composition is integrated with the Modular Continuous Learning Architecture, it enables continuous learning of skills of increasing complexity. This chapter also proposes a Kullback-Leibler divergence based metric to measure the difficulty level of a task and the agent's skill competency. These metrics enable the Modular Continuous Learning Architecture to select an appropriate task to learn given the current level of expertise of the system as a whole. That is suggested as a future task in Chapter 7.

The learned knowledge used as a building block is typically learned in the same non-scaffolded environment as the other compound skills resulting in inefficiently learned primitive skills. This thesis hypothesizes that skill composition is more effective when the primitive skills are learned in specialized/scaffolded environments. The hypothesis is based on the premise that a scaffolded environment provides the agent with a better learning opportunity. As seen in Chapter 4, some tasks were not learned due to a lack of opportunity. Thus, scaffolded environments are created for the robot to learn primitive skills. The experiments will show how the composition of primitive skills learned in a scaffolded environment can be used as a skill for a more complex task. The results of primitive skills learned in a scaffolded are compared with the same skills learned in a non-scaffolded environment to demonstrate the effectiveness of learning in a scaffolded environment. The primitive skills learned in the scaffolded environment are then combined to generate a skill for the compound tasks. The composed skill is compared with the skills learned from scratch to demonstrate the effectiveness of the proposed skill combination method. Thus, this chapter's contributions are (i) two variants of a novel ensemble method to compose policies for compound tasks that are concurrent combinations of disjoint tasks, (ii) a comparison of the performance of the skill learned in scaffolded and non-scaffolded environments and also a comparison of the skill learned from scratch with composed skill whose constituent primitive skills are learned in a scaffolded environment and (iii) a Kullback-Leibler divergence based metric to measure the task difficulty level and agent's skill competency. The rest of this chapter is organised as follows: Section 6.2 reviews the literature on skill composition. Section 6.3 proposes how primitive skills can be concurrently combined. Section 6.4 proposes the metrics to measure the task difficulty level and the agent’s skill competency. Section 6.5 describes the setup of the experiments and discusses the results. Finally, Section 6.6 provides the concluding remarks.

6.2 Skill Composition

For an open-ended lifelong learning system, a critical characteristic is that it learns skills for multiple tasks. It should be able to determine what tasks it should learn [19] [20] and when it should learn them [20]; also, to be efficient and be aware of its capabilities [57], it should be able to exploit the learned knowledge to improve the performance of the task at hand [57] [127]. As detailed in previous sections, reinforcement Learning is most suitable for open-ended learning. However, in many cases, the amount of interaction needed to learn a task is quite large [29], making it impractical for many robotics applications. While the reinforcement learning community is conscious of this drawback and is continually inventing sample efficient algorithms [41] and using different approaches such as imitation learning [33] [76], further benefits can be gained by re-using the previously learned knowledge to create a solution for the future tasks.

The reinforcement learning literature review shows several approaches to re-using previously learned knowledge to create a solution for future tasks. Since the aim is to integrate this with the continuous learning agent architecture, the scope of the review in this section is limited to approaches where the reinforcement learning policies for simpler tasks are combined to form a policy for a more complex task. In saying that, other approaches that re-use previously learned knowledge to form solutions for future related or unrelated tasks are listed to provide a broader context. Then the review focuses on techniques that combine the reinforcement learning policies to form solutions for compound tasks.

In reinforcement learning, a solution to achieve the task is called a policy. Two or more policies can then be combined to learn solutions for compound tasks. Existing literature reveals two common ways of combining reinforcement learning policies: (1) Sequentially [134] [135] – where the policies for the subtasks are invoked in a sequence to solve a more complex task. The subtasks may or may not be organised in a hierarchy; however, this technique broadly falls under hierarchical reinforcement learning. For example, consider that a mobile robot picks up an object from destination A and delivers it to destination B. Both these tasks have to be carried out one after the other in order. To form a compound skill, the skill to solve the first task is sequentially combined with the skill to solve the second task. (2) Concurrently [127] [136] – where the policies of the subtasks are merged to form a combined policy that is used to solve the complex task. Such a technique is called compositionality. Consider, for example, that the mobile robot has to follow a moving target while avoiding obstacles along the way. In that case, both primitive skills are combined in a concurrent manner to form the compound skill, i.e. they are simultaneously active. Another approach seen in the literature is called a modular combination of skills in which the skills are enabled and disabled based on a trigger. For example, consider that a mobile robot is following a track on the floor and comes across an obstacle. It navigates around the obstacle and again starts to follow the track. In this case, the robot stops using the first skill when the second skill is triggered.

First, to provide a broader context, the following subsection reviews approaches that either reuse previously learned knowledge or aim to learn a solution to compound tasks. That is followed by a more detailed review of the work that focuses on combining reinforcement learning policies.

6.2.1 Multi-task learning

In the explanation based neural network [137], Thrun uses domain knowledge to provide a context of the data for the agent to generalise the knowledge. That enables the agent to use previously learned knowledge from the *n-1* related tasks to learn the n^{th} task more efficiently. While this approach reduces the amount of training data the reinforcement learning agent requires, the approach applies only to the related tasks. The approach focuses on generalising the knowledge and applying it to newer tasks, which may or may not be more complex than the previous *n-1* tasks. Drummond [138] also uses a similar concept of transfer of knowledge from related tasks. Both the techniques mentioned above apply only to related tasks. For an open-ended learning agent, that would be considered a significant limitation.

In multi-task / multi-objective / feudal reinforcement learning [139], the agent trains to learn multiple tasks at the same time. A compound task, seen from a different viewpoint, is considered a set of multiple tasks. The multi-objective reinforcement learning agent aims to find a policy that satisfies multiple tasks instead of just one. The reward function, in this case, is a vector instead of a scalar value, and the agent aims to learn all policies. When compared to compositionality, the difference is that in the case of multi-objective reinforcement learning, the tasks are usually contradictory, and the challenge for reinforcement learning is to find a policy that optimally satisfies all the tasks. While the aim of multi-task reinforcement learning is to learn the skill to solve multiple tasks, since the skill to solve those tasks is learned as a single policy, there is no way to reuse the skill. The policy cannot be sliced and diced to generate policies for constituent tasks. Hence if one of the tasks in the set of multiple tasks were replaced with a new task, it would mean that the learning has to begin from scratch.

6.2.2 Sequential combination of policies

In this approach, the policies are combined in sequential order. That is akin to a planning problem where the previously learned policies are sequenced to accomplish a complex task. The tasks may or may not be hierarchically structured; however, the same concept of sequentially combining the policies can be applied to both. Broadly this can be considered hierarchical reinforcement learning.

Hierarchical reinforcement learning [140] aims to decompose the task into subtasks, commonly represented using options [42], learn the policies for each of the subtasks, and then treat that policy as a macro action. The solution to the complex task is a policy that sequentially invokes these macro actions. This has been an extensively researched area, ranging from auto-generation of options [141] to integrating this with the core reinforcement learning algorithm to form algorithms such as option-critic [142]. MAXQ [143] introduces mechanisms for abstraction and sharing in reinforcement learning for it to be able to solve tasks that have complex hierarchical structures. The concept exploits the regularities found when a complex task is decomposed. Such techniques of a sequential combination of skills can be integrated with the agent architecture proposed in Chapter 3.

6.2.3 Modular reinforcement learning

Modular reinforcement learning is similar to the sequential combination of reinforcement learning policies. It switches from one policy to another; however, the decision to sequence the policy is made at a run time based on the initiation trigger or termination state.

Modular reinforcement learning [144] [145] [146] [147] decomposes a task, and each module solves a portion of the task. For the final solution, at runtime, a selector then selects the policy of the subtask in sequence. Although not based on reinforcement learning, a technique worth mentioning here is the subsumption architecture [70]. It is a layered architecture where the lower layers represent more primitive behaviours, whereas higher layers represent high-level behaviours. Like modular reinforcement learning, a behaviour is triggered when a specific condition is satisfied, which is akin to switching the policy based on a trigger. The techniques that fall under the modular reinforcement learning category too can be integrated with the agent architecture proposed in Chapter 3.

6.2.4 Concurrent combination of policies

The final category is where the reinforcement learning policies are combined concurrently to form a single policy. This concept is termed as compositionality [136] [148] [149]. The policies of constituent tasks are combined, and they act in unison to form a solution for the compound task. The reinforcement learning policy can be represented as a Q-table, neural network, or basis function. When the policies are represented as a Q-table, the Q function for the compound task is generated by averaging the constituent Q functions [127] [145] [150] [151]. When the policies are represented as a neural network, the literature review shows that the combined policy is generated using voting, a Mixture of Experts [131] [132], policy distillation [152] [153] and action selection using a Gaussian Mixture Model [130]. Also, compositionality can be further enhanced by combining the concept of compositionality with modular reinforcement learning, where the final policy is generated by combining multiple policies concurrently [151].

Haarnoja et al. [150] demonstrated the combination of policies for different sets of behaviours. Using a robotic manipulator arm, it was shown how the policy to move an object on a vertical strip and the policy to move the object on a horizontal strip are combined to form a policy for the robot to be able to move the object to the intersection position of the two strips. Todorov [154] developed a theory of compositionality applicable to a general class of stochastic optimal control problems. Niekerk et al. [127] apply the concept of compositionality to the lifelong learning agent. Using a high-dimension video game use case, it is demonstrated how an agent can combine skills from its library of already learned skills to solve a new task. Niekerk et al. conclude that learning a policy for a composite task can be difficult because of the tendency to collapse into learning a single task without exploring the alternatives. Hence, it is better to learn primitive skills and then combine them to produce an optimal solution for the compound task.

6.2.5 Gap in the state-of-the-art

The literature review shows that primitive skills can be combined sequentially or concurrently for skill reuse and that the current research focus is on the former, highlighting a research opportunity. Unlike the case of the sequential combination of policies where the technique of options is commonly used, there is no common method for the concurrent combination of skills. While the concurrent composition of reinforcement learning policies has attracted attention in recent years, much remains to be done. Compositionality enables mixing and matching primitive skills to form solutions to various compound tasks. For example, when a robot learns skills A, B, C and D. Solutions to compound tasks can be composed by combining the skills such as A+B, A+B+C, A+B+D, A+C+D, A+B+C+D. That can be especially useful for the reinforcement learning robot since it may not always be possible for the robot to learn every skill from scratch. For the ones it can, such reuse of knowledge leads to sample efficiency.

The review has shown that option is a commonly used technique to store policy for sequentially combined tasks. That can be integrated with the architecture proposed in Chapter 3. However, the methods found in the literature for the concurrent combination are not task-independent or abstract enough to be easily integrated with the architecture proposed in this thesis. That raises a question: How does one design a module to concurrently compose known skills to form a solution for a compound task with the added requirement that the technique is compatible with the proposed agent architecture? The following section will aim to answer that question.

6.3 Concurrent Skill Composition

A key characteristic of a continuously learning system is learning skills for multiple tasks and using the learned skill to solve future tasks. The agent architecture proposed in Chapter 3 uses reinforcement learning to acquire skills. In reinforcement learning, the common practice is to train the agent for every task from scratch. That is not only time-consuming but also impractical, especially for many robotics applications. That is because the very nature of reinforcement learning requires the agent to find itself in a situation to be able to try different actions, and also, the agent may need to be reset to an initial/random starting state for retries [155]. That is because reinforcement learning relies on several similar learning opportunities so that the agent can explore available actions to create a mapping between states and actions, i.e. generate a policy. This sample inefficiency is a significant challenge in reinforcement learning [156]. This inadequacy of reinforcement learning becomes apparent when learning solutions to complex tasks.

With the agent architecture, such as the one proposed in Chapter 3, the system is continuously enhancing its knowledge base. This knowledge is stored in the knowledge management module of the architecture. In reinforcement learning, the solution to the task, i.e. the skill, is stored as a policy. A policy can be represented using a Q-table or using a neural network. Policies represented using Q-tables can be concurrently combined by averaging the constituent Q-values [150]. This section will detail a similar technique for the policies represented using neural networks. The technique provides the best of both worlds. It provides the simplicity and understandability of the concurrent combination used in the Q-table based representation and the neural network's scalability. This concurrent combination of the neural network skills is similar to the average model weight ensemble [36] [37].

A compound task's reinforcement learning policy can be formed by averaging the learnable parameters of the constituent tasks' networks. Starting with learning primitive tasks first, it then moves on to learning compound tasks. With a repository of primitive skills in place, it is only logical to extend the system's capabilities to reuse the learned primitive skills to form solutions for compound tasks. The advantage of this simple averaging technique becomes apparent in a multi-agent reinforcement learning setting where the agents are learning different primitive skills building a repository of skills. Those skills can then be combined as required to create a solution for a compound task.

A compound task can be considered as a sequential or concurrent combination of the constituent primitive tasks. Further, a concurrent combination can be an 'AND' or an 'OR' combination. In an 'AND' combination, for successful execution, all the constituent tasks are executed simultaneously. Whereas, in the case of an 'OR' combination, the execution is considered successful if one of the constituent tasks is executed successfully. That is, the combined policy can solve either of the constituent tasks, but not all at the same time. When learning multiple tasks, the two tasks can be said to be 'competing' if the actions required to accomplish one task are opposite to the actions necessary to accomplish another task. The tasks are said to be 'complementary' if the actions required to accomplish one task are the same as the actions necessary to accomplish another task and 'disjoint' if they are neither competing nor complementary. Like any other multi-task learning method, this approach's limitation is that the tasks should not be contradictory. Since, for such contradictory tasks, the actions may be competing in nature. The scope of this chapter's contribution is limited to the 'AND' combination of disjoint tasks.

For example, consider a set of disjoint primitive tasks for a vacuum cleaning robot: (a) detect the dirt, (b) clean the dirt, (c) avoid obstacles, and (d) detect an edge of the floor to keep the robot from falling off the stairs. A compound task with an 'AND' combination would be the combination of tasks such as (i) the robot detecting the dirt while avoiding obstacles and avoiding falling off the stairs, and (ii) the robot cleaning the dirt while avoiding obstacles and avoiding falling off the stairs. As a non-mobile robot example, consider the following hypothetical set of disjoint tasks. Consider a primitive task of juggling and another primitive task of riding a unicycle. The compound task would be to ride a unicycle while juggling. As one can imagine, this can be further extended where the compound task can be to juggle while riding a unicycle and balancing on a tight rope at the same time. Once the primitive tasks are learned, they can be considered layers/modules of the reactive architecture where the learned behaviour is executed when triggered or multiple behaviours are combined to execute a more complex behaviour.

6.3.1 Compositionality for Q-table and neural network based policy representation

Consider that an agent implemented using the agent architecture proposed in Chapter 3 has generated several potential tasks. Further, consider that the tasks are both primitive tasks that can be represented as $p_1, p_2, \dots, p_n$ and compound tasks that are combinations of those primitive tasks. Also, consider that the policy for the primitive task p_1 is represented as π_1 and the policy for the task p_2 is represented as π_2 and so on. Similarly, the policy for the compound task C is represented as π_C.

Q-table, which is one of the ways to represent a reinforcement learning policy, stores the Q-values. Those Q-values are used by the action selection strategy to select the best action in a particular state. That forms the mapping between state and action, i.e. a policy akin to a skill. The calculation of Q-values using Q-Learning is detailed in Chapter 2, Equation (2.5). It denotes how good taking a particular action in each state is, i.e. it denotes the action that should be taken in each state to maximize the cumulative reward. In simple terms, it indicates a reward that can be received if a particular action is taken, i.e. it encodes future reward. Chapter 2, Table 2.1 shows a sample Q-table. Thus, if the Q-table for every primitive task encodes the potential reward for each action in each state, the policy for the compound task, i.e. the coordination of multiple behaviours, can be generated by a summation of the constituent Q-tables [157] [127] [150], i.e. the primitive skills such as $\pi_1, \pi_2, \dots, \pi_n$ learned independently can then be combined to form a policy π_C for a compound task. The policies for individual tasks can also act as modules and then mixed and matched as required to form a skill for a compound task [145]. A normalized representation generated by averaging the constituent Q-tables is shown in Equation (6.1), where n is the number of constituent tasks.

$$Q_C(s_t, a_t) = \frac{1}{n}\sum_{i=1}^{n} Q_i(s_t, a_t) \tag{6.1}$$

The corresponding reward for the compound task C, if the reinforcement learning agent were to attempt to solve it from scratch, can be represented as an average of the rewards [158] for primitive tasks $p_1, p_2, \dots, p_n$, as shown in Equation (6.2):

$$r_C(s_t, a_t, s_{t+1}) = \frac{1}{n}\sum_{i=1}^{n} r_i(s_t, a_t, s_{t+1}) \tag{6.2}$$

where the reward for each of the primitive tasks is within the same range and $r_1(s_t, a_t, s_{t+1})$ is the reward for the task p_1, $r_2(s_t, a_t, s_{t+1})$ is the reward for the task p_2 and so on. Parameters s_t, a_t and s_{t+1} are the agent's state, the action that the agent takes in that state and the resulting state of the agent when that action is taken.

The concept of averaging the Q-tables to form solutions for compound tasks is well-researched [145] [157] [127] [150]; however, Q-table based approaches do not scale well. The other way to represent a policy is by neural networks; for example, the Deep Q Network algorithm (DQN) [159] uses neural network-based policy representation. In such algorithms, in essence, the learnable parameters of the neural network store the Q-values, which means the concept of averaging Q-tables can be extended to the policies represented by neural networks.

A concept of averaging the neural network model weights is used for supervised machine learning problems, albeit to solve the optimization challenge where the training process of neural networks fails to converge. That average model weight ensemble (Polyak Averaging) [36] technique averages the network's learnable parameters (i.e. the weights) and is shown to generate more robust solutions. In this technique, the desirable solution is achieved by averaging the weights of multiple trained models where the sets of weights from the individual model can often be noisy but averaging results in a robust solution.

Also, a similar concept of combining the learnable parameters is used in Modular Neural Networks [133] and the Mixture of Experts [160] techniques seen in general machine learning as well as reinforcement learning literature [144] [146] [147] [131] [132], where an individual network is an expert in its domain and is built and trained independently for its specific task with the final decision made based on the results of these individual networks. That allows for building a bigger network for solving compound tasks using smaller independently trained/re-trained modules resulting in quicker training than monolithic networks. However, it requires additional infrastructure such as a decision or gating network. The decision network can be implemented using some rule or could be based on voting or a mathematical operation such as summation or averaging. For example, consider the Modular Neural Networks shown in Figure 6.2. The individual modules could be trained on specific tasks such as tracking an object using a camera or detecting an obstacle using proximity sensors. The decision network can then combine the results from the individual modules to make the final decision for solving a compound task such as tracking an object while avoiding obstacles.

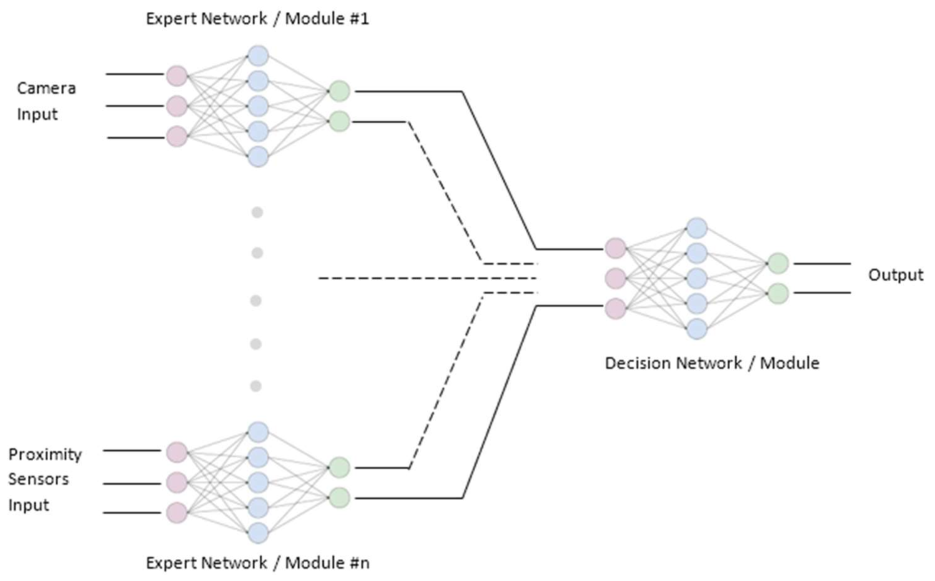

Expert Network / Module #1
Camera Input
Output
Decision Network / Module
Proximity Sensors Input
Expert Network / Module #n

Figure 6.2: An example of Modular Neural Networks

Extending the concept of averaging Q-tables and inspired by the techniques such as average model weight ensemble and Modular Neural Networks, this chapter proposes a technique to form policy for a compound task by averaging learnable parameters of policies of the constituent primitive tasks represented using neural networks. The proposed skill composition in this chapter is limited to the 'AND' combination of disjoint tasks. Since the tasks are disjoint, unlike the techniques seen in the literature [146] [131] [132], there is no overlap in state space, and hence the proposed technique does not require any additional decision/gating network infrastructure for the action arbitration in the overlapping region.

That results in a much simpler architecture for an open-ended continuous learning agent. Also, the policies for compound tasks generated using the proposed technique can be stored and recalled to be combined with additional policies compared to policies generated at run time, as is the case for modular reinforcement learning techniques. A compound task C be represented as $C = p_1 \wedge p_2 \wedge \ldots \wedge p_n$ and the policy π_C for such compound task can be represented as a combination of the policies for the constituent tasks as shown in Equation (6.3). The combined policy formed this way can be used as an initial policy that can then be refined as the agent gets an opportunity to learn more regarding a particular region of its state space.

$$\pi_C = \frac{1}{n}\sum_{i=1}^{n} \pi_i \tag{6.3}$$

6.3.2 Skill composition using average model weight ensemble

The technique proposed in this chapter is based on the actor-critic reinforcement learning algorithm [41], where the actor and the critic are implemented using neural networks. Reinforcement learning consists of algorithms that are either value-based methods where the agent learns the value function that determines how good it is to take a particular action in a specific state or policy-based methods where the agent directly optimizes the policy by sampling several rollouts of the episode. The actor-critic family of algorithms is a hybrid approach where the critic is trained to estimate the value function and provides feedback to the actor to optimize the policy. At each time step *t*, the state s_t is passed as an input to both actor and critic networks. The actor, represented as $\pi(s_t, a_t, \Theta)$, takes an action a_t in the environment receives the reward r_{t+1} and transitions to a new state s_{t+1}. Based on that, the critic, represented as $\hat{q}(s_t, a_t, w)$, assesses how good it was to take that action and accordingly adjusts the weights w of the critic network. That is then provided as feedback to the actor, resulting in the update to the weights Θ of the actor network.

Now, consider that $actor_1, critic_1$ are the actor and critic networks for the primitive task p_1; $actor_2, critic_2$ are the actor and critic networks for the task p_2 and so on. The actor and critic networks for the compound task C can be created by averaging the learnable parameters of the constituent actor networks and the constituent critic networks, as shown in Equation (6.4).

$$actor_C = \frac{1}{n}\sum_{i=1}^{n} actor_i$$

and

$$critic_C = \frac{1}{n}\sum_{i=1}^{n} critic_i \tag{6.4}$$

where $actor_c, critic_c$ are the actor and critic networks for the compound task C and can then be used to construct a reinforcement learning agent for the compound task. Typically, the networks will be multi-layer. That means the actor or the critic network of the compound task is constructed by taking an arithmetic average of the learnable parameters of the individual layers of the networks of the constituent primitive tasks. Consider the actor-network first. The learnable parameters are averaged for each layer of the actor networks of corresponding primitive tasks. Those average values are then set as the learnable parameters of the corresponding layer of actor network of the compound task. This is repeated for all the layers to construct the actor network for the compound task. The critic network is constructed by following the same process using the corresponding critic networks of the primitive tasks.

Chapter 5 showed how tasks could be generated by enabling and combining different aggregations of the state space. For disjoint tasks, these aggregations do not overlap. Hence, when the tasks are represented using the same state vector (termed representation #1 in this chapter), a compound skill can be constructed by effectively stacking the primitive skills that make up that compound task. A graphical representation of this is shown in Figure 6.3. It shows four skills, with its state space represented by a coloured and a grey area. This combination method is termed method #1 from here on in this chapter. In the diagram, the neural network based reinforcement learning policy for the skill is a combination of actor and critic neural networks. Both networks take the agent's state vector as the input. The actor network's output is the probability for each of the actions in the agent’s action space, and the output of the critic network is the feedback to the actor indicating how good it was to take a particular action in that state.

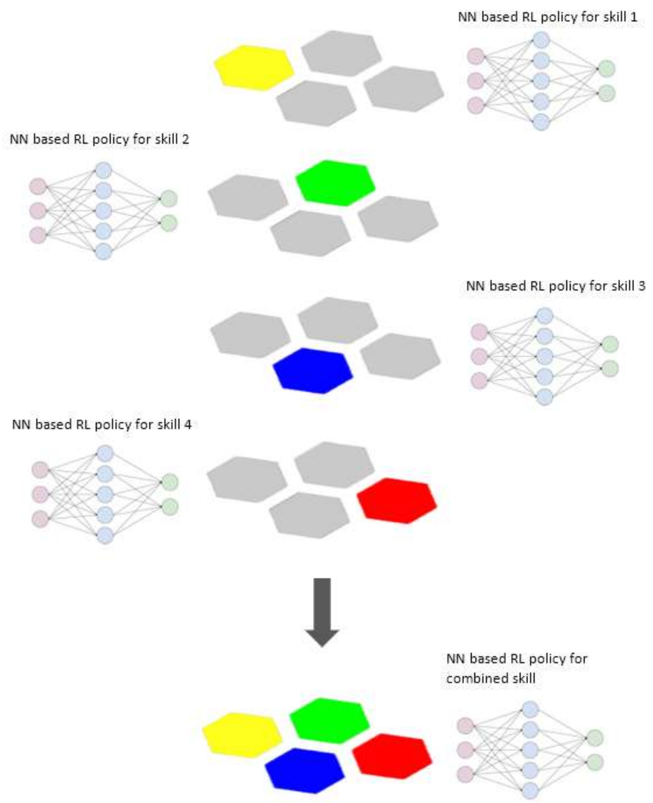

NN based RL policy for skill 1
NN based RL policy for skill 2
NN based RL policy for skill 3
NN based RL policy for skill 4
NN based RL policy for combined skill

Figure 6.3: Skill composition method #1 – same state vector for the constituent tasks (representation #1).

As an extension to the task generation technique proposed in Chapter 5, the state vector for the task can constitute just the individual aggregation (termed representation #2 in this chapter). In such a case, the state vector of each primitive task depends on the task being learned. As shown in Figure 6.4, the construction of the skill for such a compound task can be done by aligning the state vectors for the constituent primitive tasks. This mechanism is particularly beneficial in multi-agent reinforcement learning, where each agent is responsible for learning a particular skill. Those constituent skills can then be combined to form a compound skill. This combination method is termed method #2 from here on in this chapter. To learn primitive skills, the agent starts with a state space comprising only the required state attributes. Same as in method #1, the neural network based reinforcement learning policy for the skill shown is a combination of actor and critic neural networks.

However, the state-space of each task (shown using different coloured pieces) depends on the skill being learned. Reduced dimensionality leads to smaller state spaces and quicker learning. Then as required, different primitive skills are combined (by aligning to the state vector for the compound task), resulting in a skill for the compound task.

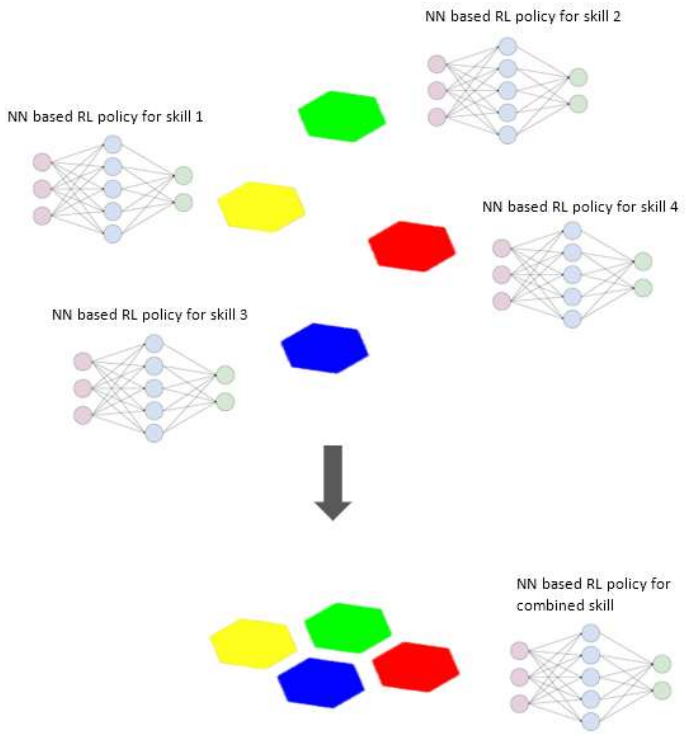

NN based RL policy for skill 2
NN based RL policy for skill 1
NN based RL policy for skill 4
NN based RL policy for skill 3
NN based RL policy for combined skill

Figure 6.4: Skill composition method #2 – task-specific state vector (representation #2).

While each composition method has its merits and demerits, both lead to reduced training time for the compound skills. That is not just because the agent is not required to learn the skill from scratch but also because different agents can learn the constituent policies in parallel. Algorithm 6.1 shows the pseudo-code for skill composition. The constituent networks' learnable parameters are averaged to construct a new combined actor-critic reinforcement learning agent that has the skill to solve the compound task. Next, this chapter proposes the task difficulty and skill competency metrics.

Algorithm 6.1: Skill Composition

Assumption

```
There is an array of learned policies for primitive skills stored in policy[]

Start

    /* Initialize variables */
    combinedActor = create actor network    // untrained actor network
    combinedCritic = create critic network    // untrained critic network
    combinedActorParams = null
    combinedCriticParams = null

    /* Ensemble of actor and critic networks */
    for i = 1: numberSkills
        actor_i = getActor(policy(i))
        actor_i_params = getLearnableParameters(actor_i)
        combinedActorParams = combinedActorParams + actor_i_params
        critic_i = getCritic(policy(i))
        critic_i_params = getLearnableParameters(critic_i)
        combinedCriticParams = combinedCriticParams + critic_i_params
    end for

    /* Calculate the average of learnable parameters */
    combinedActorParams = combinedActorParams / numberSkills
    combinedCriticParams = combinedCriticParams / numberSkills

    /* Create combined agent */
    setLearnableParameters(combinedActor, combinedActorParams)
    setLearnableParameters(combinedCritic, combinedCriticParams)
    combinedAgent = RLAgent(combinedActor, combinedCritic)

end
```

6.4 Metrology for Task Difficulty and Skill Competency

The knowledge management module detailed in Chapter 3 stores the learned skills and the metadata regarding the skills, such as the task's difficulty and the agent's skill competency for a similar task. Such metadata can be used to assess the newly discovered task's similarity (in terms of difficulty level) with the other tasks in its knowledge base. That, in turn, could be used to prioritize which task to learn. Also, it could be used to decide which primitive skills to combine to find a solution for a compound task.

This section proposes the metrics to compute the task difficulty and agent's competency for a skill for an environment with a discrete state and action space. However, the concept can be extended to an environment with a continuous state or action space. The following are the assumptions made in this section:

i. An optimal policy is available. An optimal policy is a policy where the agent executes the skill flawlessly, i.e. the agent executing the optimal policy can be said to have mastered the skill. This policy can be obtained when another reinforcement learning algorithm learns the same skill.

ii. The policy is represented using a neural network (for example, the actor-critic algorithm detailed in section 6.3).

The actor-network output is a probability distribution of the agent's actions in a particular state. This probability distribution can be compared with the probability distribution of action of an optimal policy to generate a measure of the agent's competency. Also, when the action probability distribution of the optimal policy is compared with that of a randomly acting agent, it provides a measure of task difficulty. A Kullback-Leibler (KL) divergence can be used to compare the probability distributions. The metric calculated this way does not require any task-related knowledge and is hence suitable for an autonomous agent. These metrics can also be used to derive competence-based intrinsic motivation for task prioritization and selection.

KL divergence [161] compares the difference between any two probability distributions, say *X* and *Y*. The divergence between *Y* to *X* is denoted by $D_{KL}(X||Y)$ and is computed as follows:

$$KL\big(X(\eta) \longrightarrow Y(\eta)\big) = D_{KL}(X||Y) = \sum_{a} X(\eta) log_2 \left(\frac{X(\eta)}{Y(\eta)}\right) \tag{6.5}$$

The divergence has the following properties:

- The difference is directed, which means that $D_{KL}(X||Y) \neq D_{KL}(Y||X)$.
- The difference is non-negative, i.e. $D_{KL}(X||Y) \geq 0$.
- The divergence is additive for the independent distributions. Consider X_1, Y_1, and X_2, Y_2 are independent distributions for η and κ, respectively; their joint distribution is computed as $X(\eta,\kappa) = X_1(\eta)X_2(\kappa)$ and $Y(\eta,\kappa) = Y_1(\eta)Y_2(\kappa)$. The additive property of KL divergence would mean that $D_{KL}(X||Y) = D_{KL}(X_1||Y_1) + D_{KL}(X_2||Y_2)$.

6.4.1 Metric to measure the difficulty level of a task

Task difficulty is hard to quantify; however, it can be said that accomplishing a difficult task requires more time and effort than a simpler task. A term corresponding to task difficulty in software engineering is code complexity, primarily measured using connected code paths using graph-centric approaches [162]. However, similar metrics do not exist for robotics tasks [163]. The approach in this section compares the optimal reinforcement learning policy for the task, i.e. the policy that indicates skill mastery, with that of a randomly acting agent. For a randomly acting agent, it can be assumed that the probability of taking any action would be the same.

Consider that the state space S of the agent is expressed as $\{s^1, s^2, s^3, \ldots, s^v\}$, where v is the number of states in the state space, and the action space A expressed as $\{a^1, a^2, a^3, \ldots, a^m\}$ where m is the number of actions in the action space. Also, consider that the joint probability distributions for the optimal policy and the initial policy (randomly acting agent) are denoted by X and Y. The probability distribution of actions for a randomly acting agent in the state s^i (and all the other states) will be uniform. However, when the agent has mastered the solution to accomplish the task, the probability distribution will be different for one or more states. The KL divergence for state s^i can be denoted as $D_{KL}\big(X(s^i)||Y(s^i)\big)$. Then the KL divergence for each state is computed and added to calculate the total divergence. This total divergence can be used to denote the task's difficulty level.

$$\begin{aligned} &task\_difficulty \\ &= D_{KL}\big(X(s^1)\;||\;Y(s^1)\big) + D_{KL}\big(X(s^2)\;||\;Y(s^2)\big) + \cdots + D_{KL}\big(X(s^v)\;||\;Y(s^v)\big) \end{aligned} \tag{6.6}$$

As per one of the KL divergence's properties, the divergence will be a non-negative value; thus, the total divergence will also be non-negative. It can be imagined that the greater the difficulty level of the task, the more the number of constituent KL divergences will be non-zero, thus more the total divergence.

6.4.2 Metric to measure agent's competency for a skill

Competence is defined as the quality of being capable of accomplishing a task. This section proposes a metric to calculate an agent's competency for a skill. This is done by comparing the agent's current reinforcement learning policy with an optimal policy.

Similar to the previous subsection, consider that the state space *S* of the agent is expressed as $\{s^1, s^2, s^3, \dots, s^v\}$, where v is the number of states in the state space, and the action space *A* expressed as $\{a^1, a^2, a^3, \dots, a^m\}$ where m is the number of actions in the action space. Also, consider that the probability distributions of actions for the agent's current policy and the optimal policy are denoted by *Z* and *X*. When learning to accomplish a task, the agent can be said to be aiming to emulate the optimal policy. The measure of the difference in the probability distributions across all the states provides an indication of the agent's current level of competence for the skill. The KL divergence for state s^i can be denoted as $D_{KL}\big(Z(s^i)||X(s^i)\big)$. Then the KL divergence for each state is computed and added to calculate the total divergence. Thus, the agent's competency for the skill can be represented as:

$$skill\_incompetency$$
$$= D_{KL}\big(Z(s^1)\ ||\ X(s^1)\big) + D_{KL}\big(Z(s^2)\ ||\ X(s^2)\big) + \dots + D_{KL}\big(Z(s^v)\ ||\ X(s^v)\big)$$

$$skill\_competency = task\_difficulty - skill\_incomptency \tag{6.7}$$

A randomly acting agent is considered highly incompetent as it will have a maximum total divergence value, the same as the task difficulty. As the agent masters the skill, this total divergence tends towards zero, thus increasing the measure of the agent's competency for the skill.

6.5 Mobile Robot Experiments

The previous sections proposed a skill composition technique and task-related metrics. This section will show the results of the experiments to validate the concurrent composition of skills. The experiments will use the e-puck mobile robot. The robot will learn a few primitive skills in scaffolded and non-scaffolded environments. Those skills will be combined to form a skill for a compound task. The agent performance of such combined skills will then be compared with the skills learned from scratch.

6.5.1 Objectives of the experiments

The objectives of the experiments are as follows:

- Compare the results of primitive skills learned without and with a scaffolded setup.
- Measure the effectiveness of the skill composition.
- Compare the composed skill with the skill for the compound task learned from scratch.

6.5.2 Methods and materials

The experiments in this chapter used Webots software to simulate an e-puck mobile robot and to create arenas. The agent is implemented using MATLAB Reinforcement Learning Toolbox's Advantage Actor-Critic algorithm. The specialized/scaffolded environments are created for the e-puck to learn primitive skills. Those environments are designed to provide a better opportunity for the robot to learn those primitive skills. Further, a few non-scaffolded environments are created to test and compare the skills for compound tasks learned from scratch with the skill generated by combining previously learned primitive skills.

Robot

For the experiments in this chapter, e-puck's eight proximity sensors, labelled in a clockwise direction: *Front-Right, Right-Diagonal, Right, Rear-Right, Rear-Left, Left, Left-Diagonal, Front-Left*; three ground sensors labelled *Left, Centre, Right* and camera represented using 'c' were used. The camera was to detect a specific blue robot created in some of the arenas. The experiment's state vector was represented as [p^{FR} p^{RD} p^{R} p^{RR} p^{RL} p^{L} p^{LD} p^{FL} g^{L} g^{C} g^{R} c]. Figure 6.5 is a sketched top view of an e-puck with the labelled proximity sensors, ground sensors and wheels. The action space consisted of (i) Turn Left, (ii) Step Forward, and (iii) Turn Right.

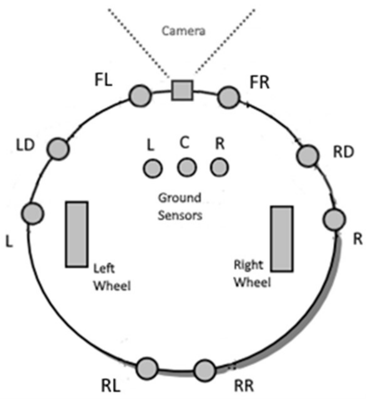


Figure 6.5: A plan view of e-puck with labelled state attributes (proximity sensors, ground sensors, wheels and camera).

State Vector:

[p^{FR} p^{RD} p^{R} p^{RR} p^{RL} p^{L} p^{LD} p^{FL} g^{L} g^{C} g^{R} c]

Actions:

{
1 - Turn Left,
2 - Step Forward,
3 - Turn Right
}

Same as the experiments in previous chapters, the proximity sensors, ground sensors and camera values were discretised into binary values. The camera was used to identify the randomly moving blue coloured robot. Webots API was used for the recognition. API returns the number of blue-coloured objects recognized in the frame. When the blue-coloured robot was in the view and recognized, the camera output was considered 1 and 0 otherwise.

Training Environment

For the experiments, several arenas were created in Webots. Figure 6.6, Figure 6.7, and Figure 6.8 show the scaffolded training arenas used to train the e-puck to learn the specific primitive skill.

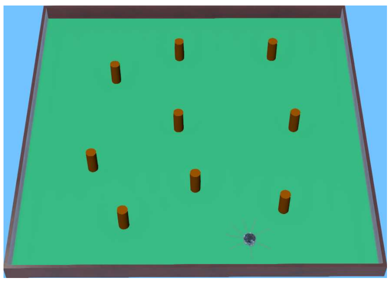


Figure 6.6: Training arena with obstacles. Specially constructed arena for scaffolded setup (Training_Arena_1).

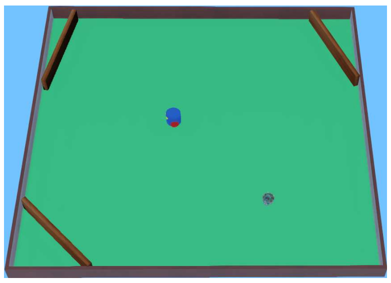


Figure 6.7: Training arena with a randomly moving blue robot. Specially constructed arena for scaffolded setup (Training_Arena_2).

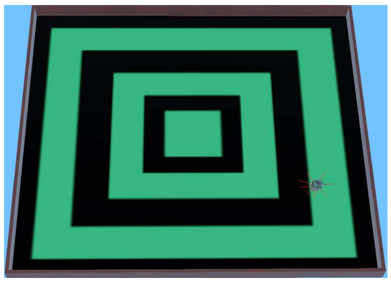


Figure 6.8: Training arena with a coloured pattern on the floor. Specially constructed arena for scaffolded setup (Training_Arena_3).

Figure 6.6 is a 2m x 2m walled arena with obstacles. In this arena, an e-puck can learn the primitive skill of avoiding obstacles/walls. Figure 6.7 is a 2m x 2m walled arena with a blue-coloured robot. This blue robot moves in a straight line, and when it detects an obstacle, it changes its direction and continues moving in a straight line. In this arena, an e-puck can learn the primitive skill of following the blue robot. Figure 6.8 is a 2m x 2m arena where the robot can learn a primitive skill of detecting the coloured floor, i.e. learn skills related to ground sensors. Such a specialized/scaffolded set of environments allows investigation of the algorithm’s performance under controlled conditions.

Test Environment

Figure 6.9, Figure 6.10, Figure 6.11, and Figure 6.12 show the 2m x 2m test/non-scaffolded arenas. Figure 6.9 is a walled arena with several black regions on the floor and a few scattered obstacles. Figure 6.10 is a walled arena with black regions on the floor and has a randomly moving blue robot. Figure 6.11 is a walled arena with a few scattered obstacles and a randomly moving blue robot. Figure 6.12 is a walled arena with obstacles, a randomly moving blue robot and black regions on the floor.

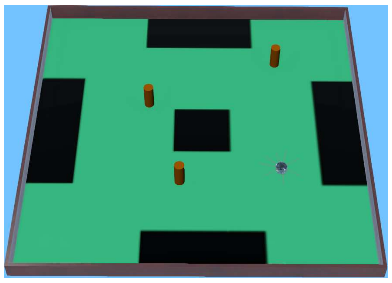


Figure 6.9: Test arena with black regions on the floor and obstacles (Test_Arena_1).

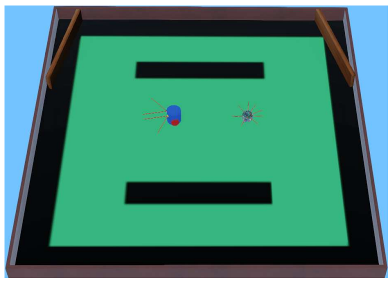


Figure 6.10: Test arena with black regions on the floor and randomly moving blue robot (Test_Arena_2).

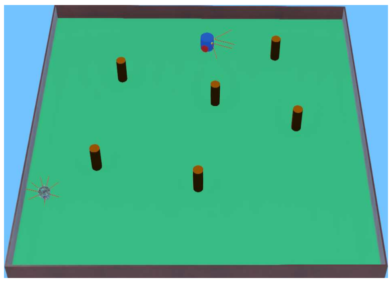


Figure 6.11: Test arena with obstacles and randomly moving blue robot (Test_Arena_3).

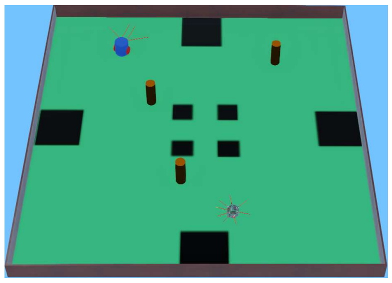


Figure 6.12: Test arena with obstacles, a randomly moving blue robot and black regions on the floor (Test_Arena_4).

Learning Algorithm

For the experiments in this chapter 6, Advantage Actor-Critic (detailed in Chapter 2) was chosen to demonstrate that the proposed ensemble technique is not merely averaging the Q-values. The algorithm implementation consists of an actor network that determines what action to take when in a particular state and the critic network that provides feedback to the actor network regarding how good it was to take that action. The agent was implemented using MATLAB's Advantage Actor-Critic algorithm from its Reinforcement Learning Toolbox. The reinforcement learning reward function for the 'maintenance' task type proposed in Chapter 4 was used to learn the solution to the tasks. The learning rate parameter was set to 0.01, and the EntropyLossWeight, the parameter that promotes exploration, was set to 0.03. The actor and the critic networks were created using the same architecture.

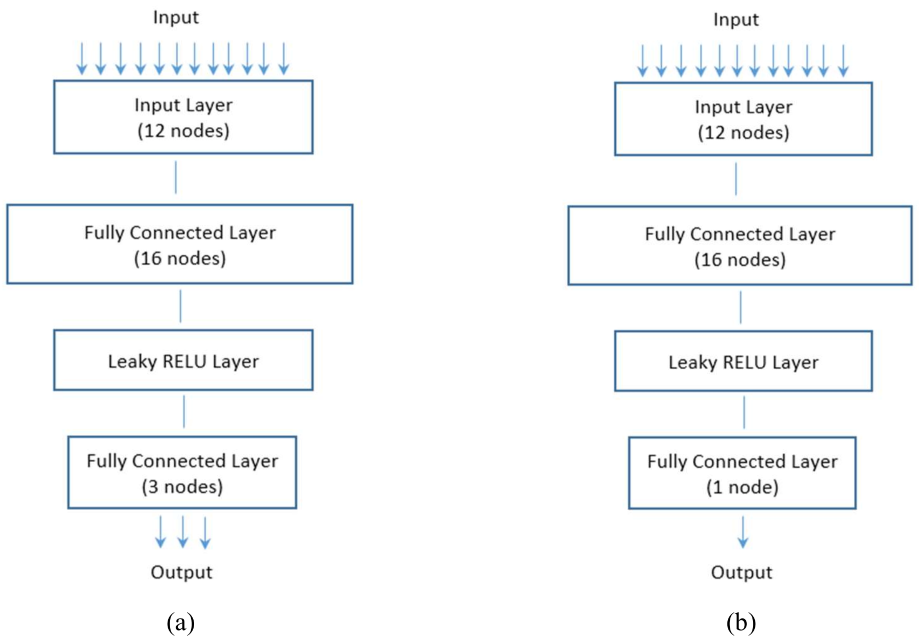

Input
Input Layer (12 nodes)
Fully Connected Layer (16 nodes)
Leaky RELU Layer
Fully Connected Layer (3 nodes)
Output
(a)
Input
Input Layer (12 nodes)
Fully Connected Layer (16 nodes)
Leaky RELU Layer
Fully Connected Layer (1 node)
Output
(b)

Figure 6.13: Neural network architecture used for the experiments in this chapter. The number of nodes in the input layer depends on attributes that make up the state; for example, representation #1 has 12 attributes. (a) The actor-network. The output is the probability for each of the three actions. (b) The critic-network. Its output is the 'advantage' calculated by the algorithm.

As shown in Figure 6.13, the networks consisted of an input layer (with the number of nodes depending on the state vector), a fully connected layer of 16 nodes, and a '*leaky RELU'* layer followed by a fully connected output layer. A non-linear activation function (leaky RELU) is used to show that the proposed technique does not just linearly aggregate the learning parameters. The number of nodes in the actor network's output layer is 3, i.e. number of actions in the action space. Each node represents the probability of that action. The number of nodes in the critic network's output layer is 1, and it outputs the 'advantage'. The' advantage' is calculated by the A2C algorithm, as detailed in Chapter 2 (Equation 2.6).

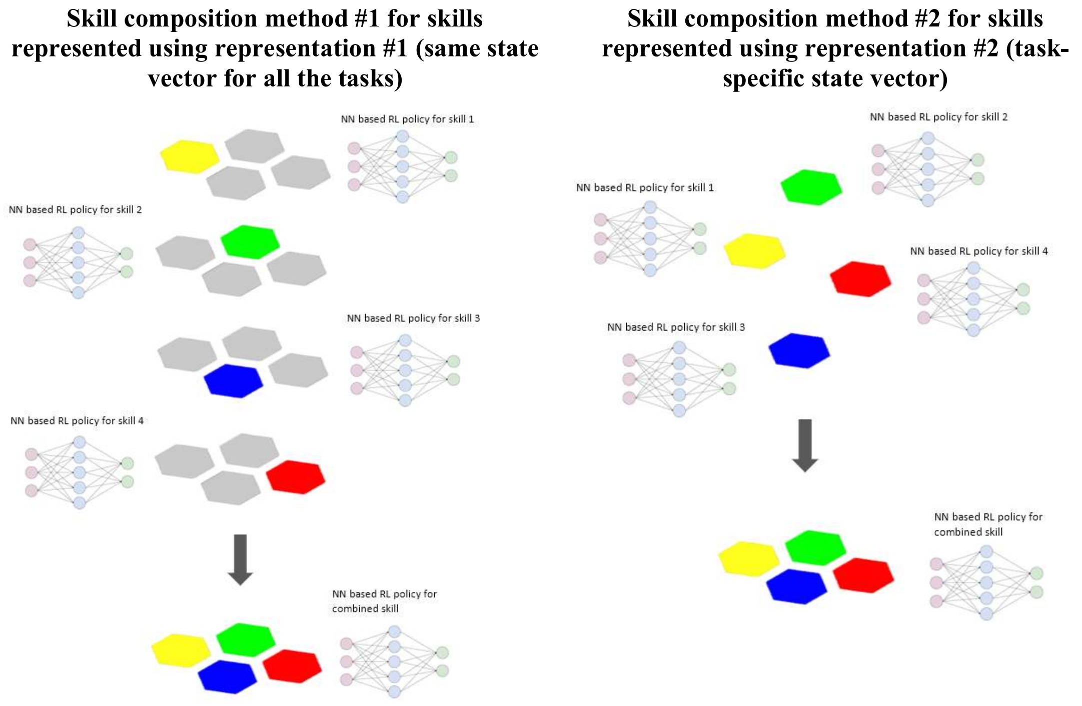
Primitive and Compound Tasks

For the experiments, primitive tasks listed in Table 6.1 were manually selected. The compound tasks were then formed by combining those primitive tasks, as shown in Table 6.1. The primitive tasks are the elemental tasks, whereas the compound tasks show the constituent tasks' composition. The column 'Task Id' shows the notation used to represent the task. Prefix 'p' is used to represent a primitive task, and 'C' is used to represent a compound task. Column 'Task Composition' details the compound task composition. The

‘Task Description’ column describes the task. ‘Arena where Trained’ column details the arena in which the training of the task took place.

Table 6.1: A list of handcrafted primitive and compound tasks that will be used for the experiments in this chapter.

Task Id	**Task Composition**	**Task Description**	**Arena where Trained**
p_1	N/A	Maintain avoiding obstacles.	Scaffolded environment – Figure 6.6 Non-scaffolded environment – Figure 6.12
p_2	N/A	Follow the randomly moving blue robot.	Scaffolded environment – Figure 6.7 Non-scaffolded environment – Figure 6.12
p_3	N/A	Maintain avoiding the black regions on the floor.	Scaffolded environment – Figure 6.8 Non-scaffolded environment – Figure 6.12
C_1	p_1 AND p_3	Maintain avoiding obstacles AND maintain avoiding the black regions on the floor.	Figure 6.9
C_2	p_2 AND p_3	Follow the blue robot AND maintain avoiding the black regions on the floor.	Figure 6.10
C_3	p_1 AND p_2	Maintain avoiding obstacles AND track the blue robot.	Figure 6.11
C_4	p_1 AND p_2 AND p_3	Maintain avoiding obstacles AND follow the blue robot AND maintain avoiding the black regions on the floor.	Figure 6.12


Skill composition method #1 for skills represented using representation #1 (same state vector for all the tasks)
NN based RL policy for skill 1
NN based RL policy for skill 2
NN based RL policy for skill 3
NN based RL policy for skill 4
NN based RL policy for combined skill
Skill composition method #2 for skills represented using representation #2 (task-specific state vector)
NN based RL policy for skill 2
NN based RL policy for skill 1
NN based RL policy for skill 4
NN based RL policy for skill 3
NN based RL policy for combined skill

State Vector for primitive tasks p_1, p_2, and p_3:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}\ g^{L}\ g^{C}\ g^{R}\ c]$

State Vector for compound tasks C_1, C_2, C_3, and C_4:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}\ g^{L}\ g^{C}\ g^{R}\ c]$

State Vector for the primitive task p_1:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}]$

State Vector for the primitive task p_2:
$[c]$

State Vector for the primitive task p_3:
$[g^{L}\ g^{C}\ g^{R}]$

State Vector for the compound task $C_1 = (p_1 + p_3)$:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}\ g^{L}\ g^{C}\ g^{R}]$

State Vector for the compound task $C_2 = (p_2 + p_3)$:
$[g^{L}\ g^{C}\ g^{R}\ c]$

State Vector for the compound task $C_3 = (p_1 + p_2)$:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}\ c]$

State Vector for the compound task $C_4 = (p_1 + p_2 + p_3)$:
$[p^{FR}\ p^{RD}\ p^{R}\ p^{RR}\ p^{RL}\ p^{L}\ p^{LD}\ p^{FL}\ g^{L}\ g^{C}\ g^{R}\ c]$

The state vectors for the primitive and compound tasks for the two ways of skill composition that this chapter proposes are shown above. The compound tasks C_1, C_2, C_3 and C_4, can be represented as follows:

$$C_1 = p_1 \wedge p_3 \qquad C_2 = p_2 \wedge p_3$$

$$C_3 = p_1 \wedge p_2 \qquad C_4 = p_1 \wedge p_2 \wedge p_3$$

6.5.3 Results and analysis

Primitive Tasks Performance

Table 6.2 shows the results of the training for primitive tasks. The training consisted of running 50 episodes of 20,000 steps each. The tasks are considered ‘maintenance’ tasks. Hence, unlike in the ‘achievement’ task type, the episode does not end once the agent reaches the desired state. So the term ‘episode’ merely means a collection of 20,000 steps. The task is learned in a scaffolded and non-scaffolded environment with the same vector for all the tasks (representation #1) and also scaffolded and non-scaffolded with a task-specific state vector (representation #2). Reward per episode was used to measure the agent performance, i.e. the average cumulative reward for the episode of 20,000 steps. A trial (50 episodes of 20,000 steps each) was run ten times for each task with different start positions of the e-puck mobile robot. Reward columns show the results averaged over those ten trials, and the standard deviation was calculated. The figure number shown in the square bracket is the training arena where the training for the primitive task took place.

The table also shows results from a statistical comparison of the training in scaffolded versus non-scaffolded environments for state vector representation #1 and scaffolded versus non-scaffolded environments for state vector representation #2. Since the results data is not normally distributed, a non-parametric method was run to compare the data points. The status quo or the Null hypothesis H_0 is that the cumulative reward received by the agent training in the non-scaffolded environment will be greater than or equal to that received by the agent training in the scaffolded environment. That is to say, a scaffolded environment does not lead to better training. Mann-Whitney U-Test was run on the 50 data points (average of ten trials for each episode) to determine if the status quo hypothesis H_0 should be rejected or not. The alpha value for this statistical test was 0.05. The Null H_0 and the alternative H_1 hypothesis can be represented as:

$$H_0: average\ cumulative\ reward\ in\ the\ non-scaffolded\ environment \\ \geqslant average\ cumulative\ reward\ in\ the\ scaffolded\ environment$$

$$H_1: average\ cumulative\ reward\ in\ the\ non-scaffolded\ environment \\ < average\ cumulative\ reward\ in\ the\ scaffolded\ environment$$

Table 6.2: Results of the learning phase for the primitive tasks. Reward per episode was measured for ten trials in the scaffolded and non-scaffolded environment and compared using Mann-Whitney U-Test.

Task Id	**Reward per episode for training in <u>scaffolded</u> environment and state vector <u>representation #1</u>.** **[Training Arena]**	**Reward per episode for training in a <u>non-scaffolded</u> environment and state vector <u>representation #1</u>.** **[Training Arena]**	**Mann-Whitney U-Test of the results shown in the previous two columns.** **Hypothesis H_0 Vs H_1?**	**Reward per episode for training in <u>scaffolded</u> environment and state vector <u>representation #2</u>.** **[Training Arena]**	**Reward per episode for training in a <u>non-scaffolded</u> environment, multi-agent learning and state vector <u>representation #2</u>.** **[Training Arena]**	**Mann-Whitney U-Test of the results shown in the previous two columns.** **Hypothesis H_0 Vs H_1?**
p_1	14816 ±3450 [Figure 6.6]	11077 ±1594 [Figure 6.12]	p-value = 0.00 Reject H_0	**15976 ±471** [Figure 6.6]	15880 ±566 [Figure 6.12]	p-value = 0.21 Reject H_0
p_2	**13117 ±2445** [Figure 6.7]	-1369 ±4366 [Figure 6.12]	p-value = 0.00 Reject H_0	11796 ±2346 [Figure 6.7]	8255 ±688 [Figure 6.12]	p-value = 0.00 Reject H_0

p_3	11799 ±5337 [Figure 6.8]	11077 ±1594 [Figure 6.12]	p-value = 0.03 Reject H_0	13206 ±4764 [Figure 6.8]	**19885 ±51** [Figure 6.12]	p-value = 1.00 Fail to Reject H_0

Table 6.2 shows a statistically significant difference in the agent's performance for the scaffolded versus non-scaffolded environment for all the tasks. Results in bold indicate the best performance for the primitive task. The Mann-Whitney U-Test results show that for tasks p_1, p_2, and p_3 tasks represented using state vector representation #1, data support the alternative hypothesis H_1 that training in a scaffolded environment leads to better learning. That is because a scaffolded environment minimizes triggering the non-skill specific sensors and allows the agent to focus on learning just one skill.

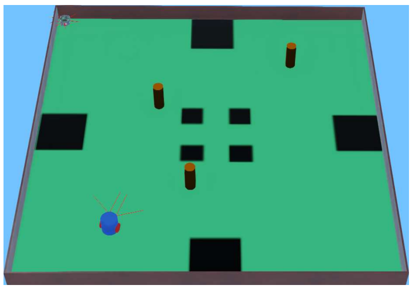


Figure 6.14: Reward hacked for task p_3. E-puck seen at the top left corner of the arena has stumbled upon a situation where it keeps pushing itself against the wall to gain a positive reward.

For tasks represented using state vector representation #2, Mann-Whitney U-Test results show that the data for p_1 and p_2 also supports the alternative hypothesis H_1, i.e. training in a scaffolded environment leads to better learning. Task p_3, where the agent performance in the non-scaffolded environment seems better is an anomaly. Upon closer examination, it was seen that it was a case where the agent had come up with an unexpected way of gaining the reward. As shown in Figure 6.14, the agent found a way to push itself against a wall while on the floor's non-black region. In this case, both wheels keep turning to move forward (and it is on the non-black region, i.e., task p_3); albeit, the wall does not allow forward motion. Once the agent stumbles upon such a situation, it exploits it by remaining in that situation. Even though the agent accumulates a high reward, it does not learn the skill to avoid black regions on the floor. That cannot happen in a scaffolded environment, and for this reason, the skills learned in the non-scaffolded environment are not used for skill composition in the next set of experiments.

In the scaffolded environment, the agent gets more opportunities to learn the skills. Although the same is true for the state vector representation #2, the advantage is not as pronounced. While the environment can trigger non-skill specific sensors on e-puck, those sensors are not part of the agent's state vector and hence do not interfere in the agent's learning of the skill in any way. For the compound task experiments in the following subsection, only the state vector representation #1 is used.

Compound Tasks Performance

Firstly, to check the composed skills' validity, the performance of the combined policies was first tested in the test arenas. Figure 6.15, Figure 6.16, and Figure 6.17 show the e-puck robot's trajectory executing the combined policies for tasks C_1, C_2, and C_3, respectively. For this visual validation, the primitive skills learned in a scaffolded environment were combined using method #1.

The combined policy for C_1 shows the behaviour of avoiding obstacles as well as the black region. Figure 6.15 shows the top view of the test arena with black regions on the floor and obstacles with the trajectory of the e-puck shown in navy colour, starting from the 'start' position. It shows that the e-puck is avoiding obstacles as well as black regions on the floor. The combined policy for C_2 shows the behaviour of the e-puck following the blue robot and avoiding the black region. Figure 6.16 shows the top view of the test arena with a randomly moving blue robot and black regions on the floor with the trajectory of the e-puck shown in navy colour starting from the position marked 'start'. It shows that the e-puck is following the blue robot, and when it reaches the black region on the floor, it changes its direction to avoid the region. The combined policy for C_3 shows the behaviour of the e-puck following the blue robot while avoiding obstacles. Figure 6.17 shows the top view of the test arena with a randomly moving blue robot and obstacles with the trajectory of the e-puck shown in navy colour, starting from the 'start' position. It shows that the e-puck is avoiding obstacles and following the blue robot at the same time.

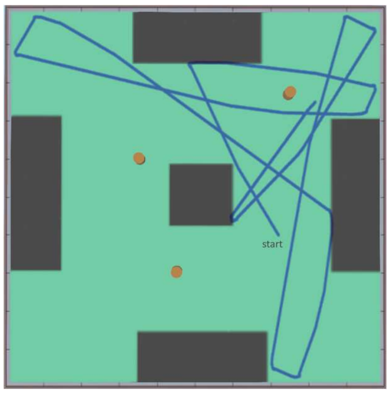


Figure 6.15: Trajectory (in navy colour) of e-puck executing the combined policy for C_1 (avoiding obstacles and avoiding black regions on the ground). The e-puck starts from the location marked 'start'.

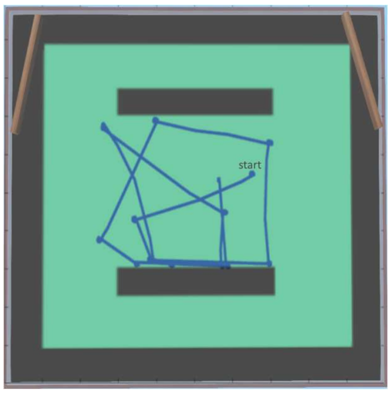


Figure 6.16: Trajectory (in navy colour) of e-puck executing the combined policy for C_2 (following a blue robot and avoiding black regions on the ground). The e-puck starts from the location marked 'start'.

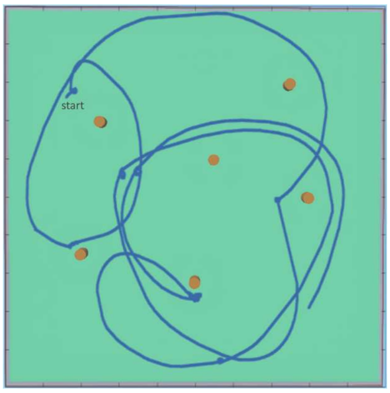


Figure 6.17: Trajectory (in navy colour) of e-puck executing the combined policy for C_3 (following a blue robot and avoiding obstacles). The e-puck starts from the location marked 'start'.

Following the skill composition validation, the training was carried out for the compound tasks C_1, C_2, C_3, and C_4. Table 6.3 shows the results of the agent learning the compound task from scratch. The column 'Task Composition' shows the compound task's composition and 'Task Description' describes the task. The training for the compound tasks constituted 50 episodes of 20,000 steps each. Ten trials were run for each task with different start positions of the e-puck mobile robot. Reward columns show the results averaged over ten trials, and the standard deviation was generated. That shows the agent's performance in attaining the respective tasks, i.e. the average cumulative reward for the episode of 20,000 steps. The experiments were run on a Dell G3 machine with Intel 10th Gen I7 6-core CPU and 16 Gb RAM. Webots was used in the 'Fast Mode' with no graphical rendering resulting in ~16x the real-time speed. The compound tasks' average learning time was approximately 35 minutes for each 50 episode run of 20,000 steps, i.e., 560 minutes if the experiment was run at the real-time speed.

The compound skills were tested for 50 episodes for this further validation, each comprising 20,000 steps. A trial for each task was run ten times with a different starting position of the e-puck. Table 6.3 shows the results for a compound skill learned from scratch and the composition of skills learned in the scaffolded environment by method #1. The table shows results for compound tasks C_1, C_2, C_3, and C_4. The last column shows the output indicating if the results for a compound skill learned from scratch and the composed skill results show a statistically significant difference. Since the results data is not normally distributed, a non-parametric method was run to determine if the data shows a statistically significant difference. The status quo or the Null hypothesis H_0 is that the reward received in the test phase by the agent using the policy learned from scratch is greater than the composed policy. That is to say, the agent using a composed policy will not perform as well as the agent using the policy learned from scratch. Mann-Whitney U-Test was run on the 50 data points (average of ten trials for each episode) to determine if that status quo hypothesis should be rejected or not. The alpha value for this statistical test was 0.05. The null H_0 and the alternative H_1 hypothesis can be represented as:

$$H_0: average\ cumulative\ reward\ for\ the\ policy\ learned\ from\ scratch$$
$$> average\ cumulative\ reward\ for\ the\ composed\ policy$$

$$H_1: average\ cumulative\ reward\ for\ policy\ learned\ from\ scratch$$
$$\leqslant average\ cumulative\ reward\ for\ the\ composed\ policy$$

Table 6.3: Results for compound tasks. Reward per episode measured for ten trials with standard deviation shown. Mann-Whitney U-Test based comparison of the skill learned from scratch and using the composed skill.

Task Id	**Task Composition**	**Reward per episode during the <u>learning phase</u> for the skills learned from scratch.** **[Training Arena]**	**Reward per episode during the <u>test phase</u> for the compound skill <u>learned from scratch</u>.** **[Tested in the arena shown in Figure 6.12]**	**Reward per episode during the <u>test phase</u> for skills learned in a scaffolded environment and <u>combined using method #1</u>.** **[Tested in the arena shown in Figure 6.12]**	**Mann-Whitney U-Test of the results shown in the previous two columns.** **Hypothesis H_0 Vs H_1?**
C_1	p_1 AND p_3	9059 ±7675	16338 ±45	15018 ±60	p-value = 1.00

		[Figure 6.9]			Fail to Reject H_0
C_2	p_2 AND p_3	-5023 ±3638 [Figure 6.10]	203 ±1537	1141 ±632	p-value = 0.00 Reject H_0
C_3	p_1 AND p_2	3875 ±1016 [Figure 6.11]	7027 ±376	4071 ±182	p-value = 1.00 Fail to Reject H_0
C_4	p_1 AND p_2 AND p_3	642 ±2990 [Figure 6.12]	1673 ±796	3065 ±702	p-value = 0.00 Reject H_0

Mann-Whitney U-Test results show that for C_1 and C_3, the data support the H_0 hypothesis, i.e., the policy learned from scratch is better than the composed policy. However, for C_2 and C_4, the test shows that the data suggests the rejection of the H_0 hypothesis, i.e., statistically, there is no difference between the policy learned from scratch and the composed policy and that the alternative hypothesis H_1 should be accepted. Figure 6.18, Figure 6.19, Figure 6.20, and Figure 6.21 show the graphical representation of Table 6.3 results. The shaded plot in red is the episode reward for the policy learned from scratch, and the shaded plot in blue is the episode reward for the agent using the combined policy (primitive skills learned in the scaffolded environment and combined using method #1).

Tasks vary in complexity, resulting in a difference in skill acquisition. Generally, the hyperparameters such as the learning rate and EntropyLossWeight are tuned to ensure optimal results or policy convergence. However, in our experiments, the hyperparameters were the same for all the tasks. That may be the reason why, in some cases, the skills of the task learned from scratch are better, and for other tasks, the composed skills are better. Regardless, the average episode reward value for the composed skills is significant for all compound tasks, which indicates that the agent is demonstrating the right behaviour. Thus, in a good case scenario, the composed skill is as good as the skill learned from scratch and in a worst-case scenario, the composed skill can be used as an initial policy that can then be refined further. In either case, the composition of skills results in time saved in learning the compound task's skill, thus demonstrating the advantage of using the proposed skill composition. When integrated with the continuous learning architecture proposed in Chapter 3, such skill composition enables the agent to learn the skills faster.

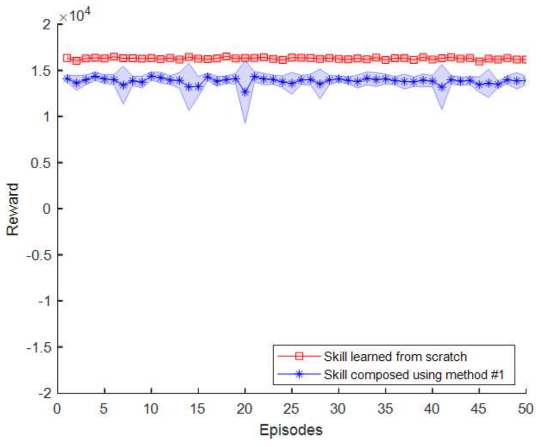


Figure 6.18: Episode reward plot for C_1 (maintain avoiding obstacles AND maintain avoiding the black regions on the floor).

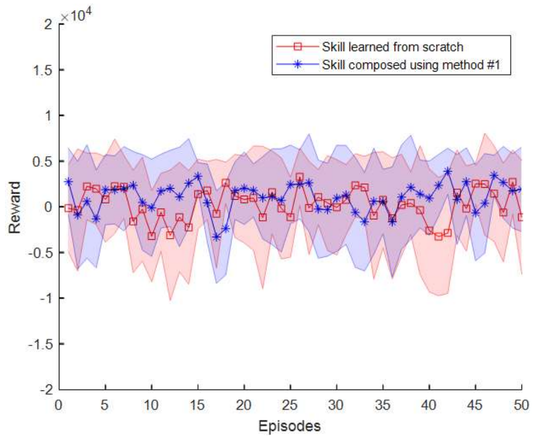


Figure 6.19: Episode reward plot for C_2 (follow the blue robot AND maintain avoiding the black regions on the floor).

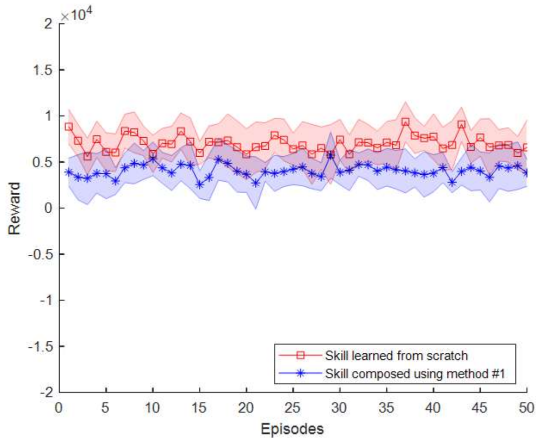


Figure 6.20: Episode reward plot for C_3 (maintain avoiding obstacles AND track the blue robot).

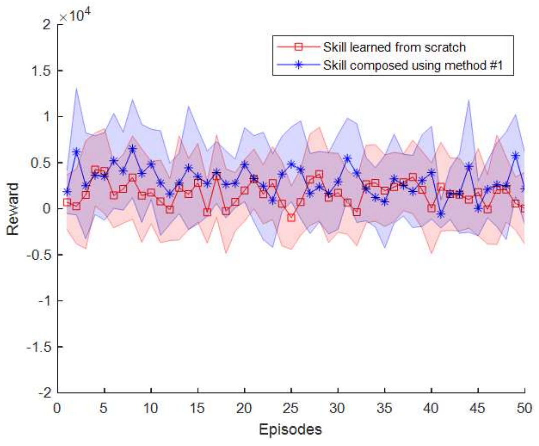


Figure 6.21: Episode reward plot for C_4 (maintain avoiding obstacles AND follow the blue robot AND maintain avoiding the black regions on the floor).

6.6 Summary

Learning every skill from scratch is time-consuming, and even with the ever-improving sample efficiency of reinforcement learning, it remains a problem, especially for robotics applications. As the continuously learning system gains new knowledge, the logical next step is to use the learned knowledge to enable faster learning of future tasks. That learned knowledge used as a building block is typically learned in the same non-scaffolded environment as the other complex skills resulting in inefficiently learned primitive skills. First of all, this chapter hypothesizes that the skill composition is more effective when the constituent skills are learned in a specialized/scaffolded environment. It compares the results of primitive skills learned in scaffolded versus non-scaffolded environments to demonstrate the effectiveness of learning in a scaffolded environment. Further, this chapter proposes two variants of a skill composition method for reinforcement learning policies represented by neural networks. It shows how the reinforcement learning policies for compound tasks can be generated by a concurrent combination of the policies for primitive disjoint tasks. Those primitive skills that are the constituent skills of the composed skill are learned in a scaffolded environment. Using mobile robot based experiments, it was shown how the combination of primitive skills could be used as a solution for a compound task with little or no additional training. A statistical comparison is used to demonstrate the effectiveness of the proposed skill combination method. In the best-case scenario, the composed policy is as good as the policy learned from scratch, and in a worst-case scenario, it can be used as an initial policy for further training. In either case, such reuse of the previously learned knowledge reduces the overall training time of multiple skills [164]. That also results in a versatile system that can mix and match the skills, an essential requirement for a continuously learning agent, especially in the robotics domain, where it may not always be feasible to learn solutions to all the tasks from scratch autonomously.

When such skill composition is integrated within the agent architecture proposed in Chapter 3, it results in an open-ended autonomous learning agent capable of continuously learning new skills. The contribution of this chapter provides a much-needed functionality to the knowledge management module of the architecture. That completes the contributions toward each of the modules of the proposed architecture. With that, the next chapter now provides the concluding remarks.

CHAPTER 7 CONCLUSION AND FUTURE WORK

7.1 Introduction

This chapter recalls the aims of this research and how this thesis fulfils the questions asked at the beginning. Chapter 1 listed the essential aspects required in the self-learning agent and raised the first question, which was “What are the modules of an open-ended and continuous reinforcement learning architecture?”. Chapter 3 proposed the agent architecture to fulfil that question. The architecture comprises a task generation module required for open-ended learning, a knowledge management module required to store the learned skills and list of tasks and a learning module implemented using reinforcement learning. The architecture allowed flexibility in terms of the addition of modules and techniques used to implement each of the modules. The second question asked was, “How does one design a module to generate task-independent reward functions for different types of tasks, including when the primitive tasks are combined to form a compound task?”. Those reward functions were proposed in Chapter 4. It meant that once the task is assumed/determined to be of a specific type, a reward function for that task type can be used regardless of the task, requiring no external intervention during the learning phase. The third question asked was, “How does one design a module to self-generate tasks of varying complexity?”. Chapter 5 proposed a task generation technique using agglomerative hierarchical clustering to generate tasks ranging in complexity from simple to complex tasks. Finally, the fourth research question was, “How does one design a module to compose a skill for a compound task by combining primitive skills?”. Chapter 6 proposed an average model weight ensemble technique of combining primitive skills to solve a compound task. Thus, with all the four questions fulfilled, this chapter now summarises the contributions of this thesis.

The rest of this chapter is organised as follows. Section 7.2 summarises the contributions of this thesis, maps them back to the research questions and details how they fulfil the questions. Section 7.3 lists the future directions for this research, and finally, Section 7.4 provides concluding remarks.

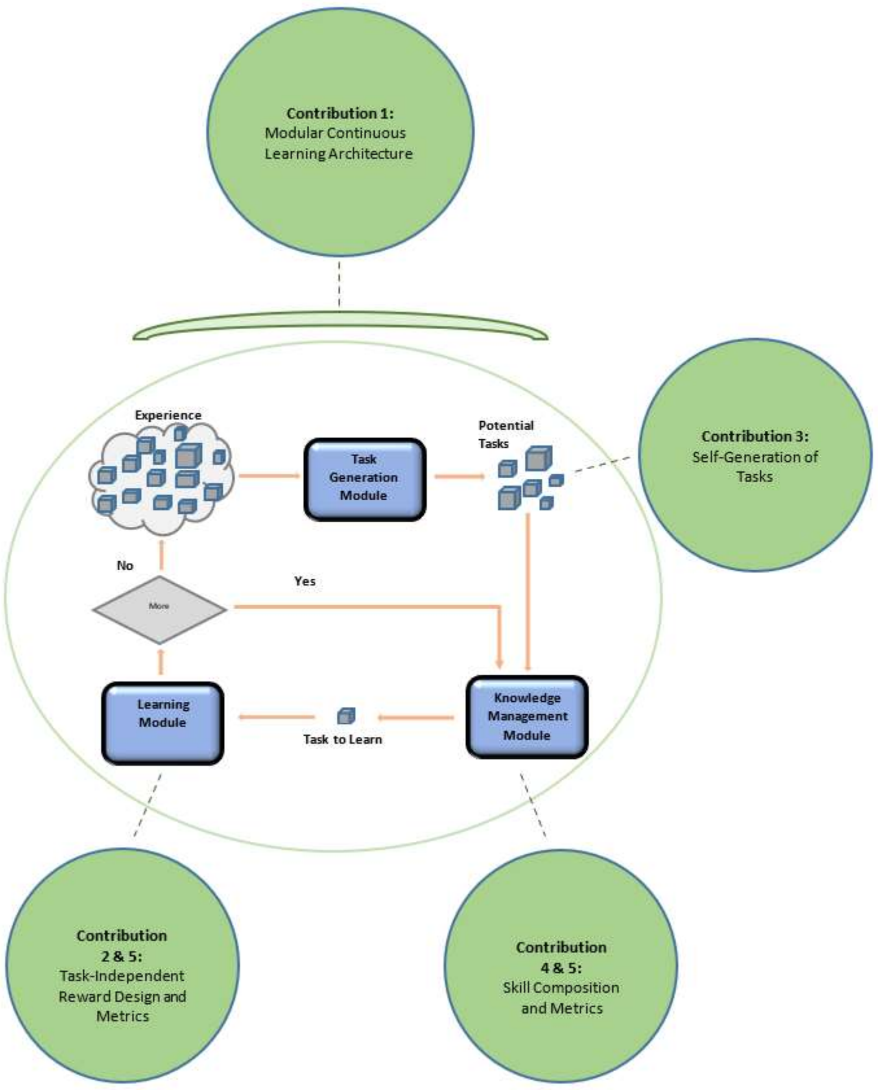


Figure 7.1: A graphical view of how the proposed architecture relates to the other contributions of this thesis.

7.2 Summary of Contributions

This section will summarise the contributions made by this thesis and show how they fulfil the research questions listed in Chapter 1. Firstly, see Figure 7.1, which shows a graphical view of the contributions of this thesis.

7.2.1 A modular agent architecture for open-ended continuous reinforcement learning

Chapter 1 identified that the key aspects for a self-learning agent are that it should: (i) be able to learn in an open-ended manner, which provides the direction to its learning; (ii) be able to assimilate the new knowledge to continuously improve its skills and be able to reuse that learned knowledge to solve future tasks; and (iii) be able to learn with minimal external intervention. With that in mind, the first research question was: what are the modules of an open-ended and continuous reinforcement learning architecture?

A review of the literature showed that architectures such as the ones proposed by Santucci et al. [12] and Merrick et al. [11] focus on open-ended learning aspects such as what the agent should learn next. Then, architectures such as [50] [165] focus on lifelong learning aspects such as knowledge assimilation and its reuse to solve future tasks. However, neither of those architectures fulfils all the key aspects listed above. The architectures that focus on open-ended learning emphasize the importance of task generation to direct the learning; however, it lacks the module that assimilates the new knowledge and its usage to solve future tasks. The architectures that focus on lifelong learning emphasize the importance of the reuse of learned knowledge; however, they lack the module that generates a constant stream of tasks to direct the learning.

This thesis proposed a modular learning architecture detailed in Chapter 3. It comprises (i) a task generation module fulfilling the criteria of making the architecture capable of open-ended learning, (ii) a knowledge repository that stores learned skills, fulfilling the criteria of making the architecture capable of continuous learning, and (iii) a learning module implemented using reinforcement learning fulfilling the requirement of learning by interacting with its environment with minimal external intervention. This flexible architecture interoperates with task generation techniques such as SART-based clustering [38] or novelty-based methods [69], as seen in the literature. The skill storage can be implemented in different ways, for example, using the option, Q-table or neural network-based policy representation seen in the reinforcement learning literature. Using an e-puck mobile robot based experiments, Chapter 3 showed how the robot self-generates its tasks, learns the skills, and continues that throughout its lifetime, thus gradually improving its capabilities in an open-ended and continuous manner. Therefore, the main modules of the proposed architecture are the modules required for an open-ended and continuous reinforcement learning architecture, thus fulfilling the first research question.

Also, as seen from the review of the motivated reinforcement learning literature, the focus of those architectures is on learning autonomy, and the focus of goal-oriented agent architectures is on generating tasks to direct the agent's learning. Thus, neither of them forms open-ended continuous learning agent architecture. The architecture proposed in Chapter 3 has a task generation module, and its learning module can be implemented using motivated reinforcement learning, thus creating a motivated reinforcement learning agent architecture with the capability to learn in an open-ended and continuous manner. Therefore this contribution extends the literature on motivated reinforcement learning agent architecture.

7.2.2 Task-independent reward functions based on the type of task

The learning in the case of a self-learning agent should be autonomous, i.e., with no external intervention. Considering that, at the outset, the learning framework chosen for the agent architecture was reinforcement learning, where the agent learns by directly interacting with its environment. In reinforcement learning, the learning is driven by reward, which in most cases are human-designed. For open-ended learning, the tasks that the agent will require to learn are not known upfront; hence it is not possible to design the reward functions in advance. Thus, for open-ended learning, reward design is an important consideration. That led to the second question: how does one design a module to generate task-independent reward functions for different types of tasks, including when the primitive tasks are combined to form a compound task?

A review of the literature showed that intrinsic motivation is commonly used to generate a task-independent reward function [83] [45] [44]. Intrinsic motivation is either knowledge-based, i.e., based on some form of prediction error or competence-based, i.e., based on the current competency to solve the task [85]. In either case, while the reward function based on intrinsic motivation is task-independent, additional domain-specific information is required to integrate intrinsic motivation into open-ended learning agent architecture. For instance, for a knowledge-based intrinsic motivation using novelty, one needs to specify what constitutes novelty.

As an alternative, Chapter 4 proposed a reinforcement learning reward design based on categorising how the tasks are considered to be attained. The common categories of such categorisation are achievement, maintenance, avoidance and approach [32]. Since the reward design based on such categorisation exploits the inherent property of how the tasks are attained, it makes them task-independent. Chapter 4 proposed the reward design for the maintenance, achievement, approach and avoidance types. This is the first time a broad spectrum of task types has been considered in a motivated reinforcement learning setting. Metrics were also designed to measure how well agents complete these tasks. The experiments in Chapter 4 showed how the robot using the proposed reward functions learns to attain tasks of different types. It also showed that such reward design could be extended to be used for compound tasks—thus fulfilling the second research question.

A reward is considered task-independent when the same reward function can be used for different tasks. For example, since the proposed reward design is based on the inherent property of the task's type, the same reward can be used for all the tasks considered as achievement tasks making the reward design task-independent. On similar lines, there is another concept called the domain-independent reward function. For example, consider a task of 'achievement' type from the games domain where the aim of the character is to navigate through a maze to reach the target and robotics domain where the aim of the robot is to pick and place an object. Since the proposed reward design is based on the task type, the same reward function can be used. That means the proposed reward design is domain-independent as well. This reward design based on task type adds to the repository of the various ways in which autonomous reward functions can be designed, extending the literature on reinforcement learning reward design.

7.2.3 A technique to self-generate tasks of varying levels of complexity

Open-ended learning, one of the key aspects of a self-learning agent, directs the learning of the agent. That direction depends on the tasks that the agent has to learn, and in many cases, those tasks cannot be predetermined. Thus, the agent should be able to design its own task set, starting with learning relatively simpler tasks first and then progressing on to learning more complex tasks. That led to the third question: how does one design a module to self-generate tasks of varying complexity?

A review of the literature showed that the task generation techniques either generate single level tasks [12] [38] [63] [69] or generate sub-tasks given an over-arching task [112] [115]. While any of these task generation techniques can be used with the agent architecture proposed in this thesis, the review highlights a gap in the literature. There are no techniques that can generate tasks of varying complexity.

Chapter 5 proposed a task generation technique based on agglomerative hierarchical clustering with that research opportunity in mind. The clustering generates regions within the agent's state space. The size of regions or aggregations can be varied by adjusting the number of generated clusters, resulting in fewer clusters, i.e. coarser aggregations, to more clusters, i.e. granular aggregations. These aggregations can then be enabled and combined to generate tasks of varying complexity, ranging from simple to complex. Using simulated e-puck mobile robot experiments, Chapter 5 showed how the robot self-generates the tasks of varying complexity—thus fulfilling the third research question.

Also, the proposed technique is designed so that as newer aggregations become available, they can be integrated within the existing unique list of aggregations, which can be used to generate new tasks. That means that newer tasks of varying complexity can be generated without making the previously generated tasks obsolete, thus enabling directed and continuous learning. As mentioned above, the literature review on task generation showed that existing techniques fall under one of the two categories: (i) generating single level tasks and (ii) generating sub-tasks given an overarching task. Since the proposed technique does not fall under either of those, it extends the literature on the auto-generation of tasks.

7.2.4 A technique to concurrently compose primitive skills to form solutions for compound tasks

Most real-world tasks are compound tasks. A compound task can be composed of primitive tasks that are sequenced together or are concurrently combined. It is a logical next step for a continuously learning agent to reuse the learned primitive skills to solve compound tasks.

Such skill reuse saves time and is also a solution to the problem that learning every skill from scratch may not always be possible. While there are solutions found in the literature for sequentially combined tasks [43], the concurrent combination of tasks lacks a similar level of research. That led to the fourth question: How does one design a module to compose a skill for a compound task by combining primitive skills?

A review of the literature showed that a reinforcement learning policy for the compound task that is a concurrent combination of constituent tasks could be created by averaging the Q-values of the constituent policies [127] [145] [150] [151]. The Q-table based approaches, however, are not scalable. Recently, several techniques have been proposed for the policies represented by a neural network. In those techniques, the combined policy is generated using techniques such as voting, a mixture of experts and action selection using a mixture model [136] [131] [130].

First of all, Chapter 6 proved the hypothesis that learning in a specialized/scaffolded environment is more efficient compared to a non-scaffolded environment. The results for primitive skills learned in the scaffolded were compared with the skills learned in the non-scaffolded environment to prove the hypothesis. Then, the chapter proposed two variants of a skill composition method for reinforcement learning policies represented by neural networks. It was shown how the reinforcement learning policies for compound tasks could be generated by a concurrent combination of the policies for primitive disjoint tasks. Using a mobile robot-based experiment, it was shown how the combination of primitive skills could be used as a solution for a compound task requiring little or no additional training. This results in a composed policy that is as good as the policy learned from scratch in the best-case scenario and a policy that can be considered a good initial policy in the worst-case scenario. In either case, such reuse of the previously learned knowledge reduces the overall training time of multiple skills. That also results in a versatile system that can mix and match the skills, an essential requirement for a lifelong learning agent, especially in the robotics domain, where it may not always be feasible to learn solutions to all the tasks from scratch autonomously—thus, fulfilling the fourth research question.

Also, the proposed technique provides an alternative to the concurrent skill composition techniques found in the literature. It offers the simplicity of combining Q-table based policies while maintaining the scalability provided by the neural network based policies. Thus, contributing to the literature on skill composition.

7.2.5 Metrology for agent performance, task difficulty and agent competency

In the case of reinforcement learning, the commonly used metric to measure an agent's performance is the reward gained by the agent in each episode. That metric, however, is only applicable to achievement type tasks, i.e. when the desired state is reached, the episode is considered to be completed, which can be used to calculate the cumulative reward received during that episode. For the tasks of other types, such as maintenance, avoidance and approach, that metric cannot be used. For instance, maintenance tasks are non-ending; thus, the concept of the episode is not relevant. Chapter 4 proposed metrics such as the regain attempts, a critical measure for non-episodic tasks.

Also, Chapter 6 proposes a metric to measure the difficulty level of a task and the agent's competency for a skill. These are useful for a continuous learning agent that always has a list of tasks waiting to be learned and needs to prioritize the skills that it can aim to learn. Not all tasks are learnable. Also, there will be tasks that are not learnable, given the agent's current knowledge level. With sufficient training on similar tasks, those previously unlearnable tasks may become learnable. Thus, the need for a metric to measure the task difficulty and agent competency. Using Kullback-Leibler divergence as a measure, Chapter 6 proposed metrics to measure these. Thus, contributing to the literature on metrics.

7.3 Future Work

While this thesis has proposed an agent architecture and made contributions towards open-ended and continuous learning, that results in only a small step towards creating a truly autonomous agent. This section lists the future directions directly related to this thesis's topic and for which it provides stepping stones.

7.3.1 Long term autonomy for a robot in real-world – overcoming the limitations of the proposed architecture

The experiments in this thesis are based on the e-puck mobile robot. By design, all the concepts of the agent architecture proposed in Chapter 3 are general enough to be compatible with other robot types. However, this has not been examined in this thesis. Also, in the experiments in this thesis, discrete state and action space was used, and the mobile robot's sensor values were discretised to binary values. The architecture or its modules or the reinforcement learning algorithm used did not dictate that but was done to keep the experiments focused on validating the core claim of the contributions. As a first step, the agent architecture detailed in this thesis should be used on another mobile robot model, using finer discretisation or continuous state/action space. That should be followed by its application to other robots, such as humanoid or animal-shaped robots or robotic arms, which may require robot specific implementation of the individual modules of the architecture. Following this, to validate the open-ended and continuous learning aspects that the architecture provides, it should be tried on service robots (wheeled/non-wheeled) that are expected to carry out real-world tasks, keeping in mind that the architecture only provides the capability to learn perception-action skills and not language-based communication skills.

An ultimate aim of autonomous agent research is to create agents that can function autonomously forever, i.e. the focus is on executing the tasks and not on the learning aspect. The architecture proposed in Chapter 3 focuses on developmental learning, such as determining what to learn, when to learn, and storage and recall of skills. That aspect would help when the sensors, actuators, and features are upgraded over the robot's lifetime. While the factory settings would provide such a robot with some skills, others would have to be learned 'on the job' when required, taking into consideration the enhanced capabilities of the upgraded software and hardware. However, even for such learning-focused architecture, there are other aspects to consider for real-world robotic applications capable of long-term autonomy. For example, (i) for the robot's tasks, that learning has to be carried out safely, i.e., without any damage to itself or its environment [166], and (ii) there has to be a smooth transition from one task to the next [135], as it is not always possible to reset the robot back to a known starting state before the next task is executed. Such seemingly minor ‘engineering’ issues have to be factored in the robot’s design for it to learn in a real-world setting.

7.3.2 Sequential combination of tasks

Tasks in the real world are generally compound tasks, and even a task that is defined as a single task can be broken down into subtasks. For example, consider a mobile robot task of “follow the wall to the left” that was considered in Chapter 4. As shown in that chapter, the task can be broken down into a series of situations, such as the robot learning to negotiate a concave corner, a convex corner, and a straight stretch, to name a few. The complexity of that task and the skills that the robot would require depends on the wall’s contour, i.e. its environment. It may not be possible to envisage in advance all the possible contour designs and hence not possible for the task designer to design upfront the wall following skills that the robot will require. In that chapter, hand-crafted if-then-else rules were used to solve a compound task that is a sequential combination of primitive tasks. Results showed how the agent learns the compound task quite quickly compared to learning such a task from scratch. So is it possible for an autonomous agent to self-generate such rules?

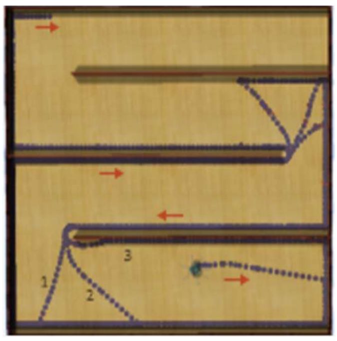


Figure 7.2: Trajectory (shown in blue colour) of the e-puck learning to follow the wall. The red arrows show the direction of the path. This figure is the same as Figure 4.11(c). It is repeated here for convenience.

To answer the question, recall the hand-crafted if-then-else rule for the wall following task in the maze arena and the resulting trajectory from that experiment, shown in Figure 7.2 for a quick reference. If the agent was to learn that wall following task from scratch, typically, reinforcement learning will take a long time. However, once the primitive tasks are learned, they can be combined using the if-then-else rules, resulting in much quicker learning. As elaborated in Chapter 6, such reuse of learned knowledge is essential for a continuous learning agent. One potential solution could be to treat this as a planning problem. Once the tasks are planned, they are executed in sequential order. However, such a solution requires a planning module in the agent architecture. Another solution is to adapt the layered control system (subsumption architecture) proposed by Rodney Brooks [70]. In both these cases, the reinforcement learning policy can be packaged as an option with a trigger condition and an end state.

7.3.3 Self-generation of high-level achievement tasks

Chapter 2 mentioned that one of the task categorisations is based on the functional aspect of how the task is accomplished, resulting in tasks of achievement, maintenance, avoidance and approach type. Also, another categorisation of tasks is based on whether the tasks are low-level, relating to actuators of a robot's joints or high-level, relating to the robot's behaviour that can be identified. In the experiments in Chapter 4, the achievement tasks were generated by the definition of an event represented by Equation 7.1. The event, as detailed in Chapter 2, models the transition between the states.

$$E_t = s_t - s_{t-1} \tag{7.1}$$

An action taken by a reinforcement learning agent can cause a transition, and an event is used to represent that transition. However, this definition of the transition considers the transition of a one-time step, and Merrick et al. [38] use that to generate achievement tasks. Such achievement tasks can be seen as low-level tasks. So, that raises a question. Can this concept of event be extended to generate high-level achievement tasks?

A potential solution can be to extend the event transition to encompass *n* time steps instead of just one. In such a case, the event/achievement task is the whole trajectory as opposed to the 'task' or the 'goal' being a single state that has to be achieved. That results in a high-level achievement task. An event in such a case can be represented, as shown in Equation 7.2.

$$E_t = s_t - s_{t-n} \tag{7.2}$$

Take, for example, an achievement task of a robot gripping a bottle or opening the lid of a bottle or a Nao humanoid robot learning to walk or wave. In order to attain such tasks, the robot has to learn to execute the steps in a particular order. Only then does it result in the task being achieved. The experiments in this thesis used the e-puck mobile robot. It would be interesting to use a different type of robot, for example, the Nao robot, to self-generate such high-level achievement tasks as follow-on research.

7.3.4 Task prioritization using agent's general competency

Using the architecture proposed in Chapter 3, the agent constantly increases its overall knowledge of the environment, improving its general competency from being a novice to an expert. The open-ended learning architecture can provide a list of tasks; however, if that list is not an ordered list, the learning would be unstructured, thus making this journey to becoming an expert longer than it should be. That raises the following questions: (i) For a continuous learning agent, how does one determine the agent's current level of general competency? (ii) Is it possible to order the list of tasks to be learned based on the agent's current level of general competency?

The solution to the second question can be to use the agent's intrinsic motivation to order the list of tasks. As seen in Chapter 3, the agent architecture proposed by Santucci et al. [12] used the motivation signal based on the learning progress to select a task from the list of tasks. Competence is one of the main motivating factors for humans to carry out a task [167], and the perceived challenge can be used to derive intrinsic motivation [95]. These concepts can be extended to the continuous learning agent and implemented in the knowledge management module of the architecture proposed in Chapter 3. As the agent progresses from a beginner to an intermediate level to an expert level, the tasks it perceives as challenging change. Based on the challenge-point hypothesis [168], tasks that lead to an optimal challenge result in the highest motivation signal. Those tasks can be prioritized for learning at that point in time.

Regarding the first question, it is difficult to quantify the agent's general competency. Such a metric will also enable one to determine if the agent is a beginner or an expert. It would be interesting to pursue such quantification as follow-on research.

7.4 Concluding Remarks

At the very start, based on the review of the literature, this thesis stated that the key aspects of the architecture to create an autonomous agent are open-ended learning, with its emphasis on a meta-cognitive aspect such as ‘what to learn’, continuous learning with its emphasis on the cognitive aspect of assimilation of the new knowledge and reinforcement learning providing an interactive solution for ‘how to learn’. This research proposed a ‘Modular Continuous Learning Architecture’ that has the components that satisfy those aspects, namely a task generation module that provides the direction for learning, a knowledge management module that is a skills repository and a learning module implemented using reinforcement learning. All these components were put into action and demonstrated using an e-puck mobile robot to show how the agent starts with having no knowledge of its environment, continuously learns in an open-ended manner, and autonomously increases its overall knowledge. Though the proposed modular architecture can interoperate with existing techniques, the review of the literature showed: (i) a lack of task-independent reward design for the agent to operate autonomously, (ii) a lack of a mechanism to generate tasks of varying complexity for it to continuously increase its overall knowledge by learning simpler tasks and then progressing on to learning more complex tasks and (iii) lack of technique for an agent to reuse its learned knowledge to solve compound tasks that are a concurrent combination of its constituent tasks. This thesis made contributions in each of those areas by proposing a reinforcement learning reward design based on the type of the task, a task generation technique for generating tasks of varying complexity and a skill composition technique that reuses learned knowledge to solve tasks that are a concurrent combination of simpler tasks. Although much remains to be done, the contributions of this research fill some gaps found in the literature, enabling one to take a step further to create a truly autonomous and self-learning agent.

APPENDIX A: SURVEY OF MOTIVATED REINFORCEMENT LEARNING ARCHITECTURES

Chapter 3, Section 3.2 reviewed the different motivated reinforcement learning and goal-oriented agent architectures. This appendix section continues that review. It does a side by side comparison of those architectures and lists the metrics used to measure the agent performance.

A.1 Comparison of Motivated Reinforcement Learning Agent Architectures

Table A.1 compares the motivated reinforcement learning architectures seen in the literature. The comparison is based on the agent's overall learning rate implemented using the architecture, the agent's learning efficiency, and how scalable the architecture is in terms of recall of learned skills. It also lists the publications that use that architecture.

Table A.1: Comparison of motivated reinforcement learning agent architectures.

	Agent Learning Rate	**Agent Learning Efficiency**	**Scalability of Skill Recall**	**Publications that Use the Architecture**
Motivated Flat Reinforcement Learning	The learning rate of MFRL is significantly faster increases compared to other models. The policy adapts to represent the task to be learned.	The learning policy cannot be recalled; hence the policy will have to be re-learned if the task becomes motivating on more than one occasion.	The learning policy cannot be recalled	[45] [44] [169] [170] [171] [172] [173] [174] [175] [176] [177] [178] [49] [179] [180] [181] [163] [182] [91] [183]
Motivated Multi-Option Reinforcement Learning	Because MMORL learns multiple skills, the learning rate appears slow; however, it is still faster than having to learn a single goal N number of times, as is the case with MFRL.	The learning policy can be recalled.	MMORL is more scalable than MHRL in terms of recalling the options.	[44] [184] [185] [186] [187]
Motivated Hierarchical	Because of the ability to reuse the recalled behaviour, the learning	The learning policy can be recalled and reused.	The skills are arranged in a hierarchical fashion;	[188] [49] [44] [189] [190] [187] [191]

Reinforcement Learning	rate can be sped up if the appropriate options are chosen.		hence it is not as scalable as MMORL.	[192] [115] [98] [193] [194] [96] [195]
Motivated Introspective Reinforcement Learning	The MIRL is able to learn quicker than the motivated RL agent without introspection.	In comparison to the architectures that lack introspection, the agent with introspection is better able to focus on complex goals.	The MIRL agents can store and recall partially learned solutions. Thus those skills do not need to be relearned from scratch. That makes the recall more scalable.	[11]

From this table, several observations arise. First, all the architectures except the motivated introspective learning architecture seem to be derived from motivated flat reinforcement learning. The main difference is the type of reinforcement learning algorithm used in architecture. The key attribute that differentiates the architectures is options, which is primarily a reinforcement learning construct. So far, there have been no attempts to design architectures that distinguish the way the key intrinsically motivated reinforcement learning attribute, namely motivation, is incorporated into the architecture. Different architectures could be formed using the way the motivation signal is either combined with reward or used exclusively. Also, it can be seen from the review of the architectures that most of the literature seems to use simpler architectures like motivated flat reinforcement learning and motivated hierarchical reinforcement learning. Even though architectures like motivated multi-option reinforcement learning and motivated introspective reinforcement learning appear to be more comprehensive and capable of representing and learning complex sets of skills, they seem to be seldom used. In addition, no architectures are currently proposed to combine motivation with other types of reinforcement learning algorithms, such as multi-agent reinforcement learning and multi-objective reinforcement learning, although motivation has recently been examined in a deep reinforcement learning setting [96][97]. These observations show that although intrinsically motivated reinforcement learning is a promising framework, much remains to be done as far as the architectures are concerned.

A.2 Performance Measures for Agents

One of the main reasons for combining intrinsic motivation with reinforcement learning is to be able to create an agent that can learn a different skill with little or no external intervention. That raises a question regarding the evaluation of such an agent's performance compared to a simple reinforcement learning agent. This section reviews the performance metrics used in the intrinsically motivated reinforcement learning literature to compare different aspects of agent performance.

Macindoe et al. [163] proposed learning efficiency, behavioural variety, and complexity as metrics to measure the performance of intrinsic motivation. The learning efficiency was calculated by using the number of steps taken by the agent to converge to a stable policy. Behavioural variety was calculated using the number of tasks for which a stable policy was learned, and behavioural complexity was calculated using the total number of actions required to accomplish a task. Merrick and Maher [173] generated behaviour patterns in non-player game characters to measure the effectiveness of intrinsic motivation. This was then used to show that these non-player game characters adapt better to the game's changing environment. Merrick et al. [196] extended this further to a behaviour structure concept that is then used to evaluate the emergent behaviour of a motivated reinforcement learning agent.

As an alternative, as shown by Barto et al. [188] [184], one of the ways to compare the performance of an intrinsically motivated reinforcement learning agent to a reinforcement learning agent is to use an intrinsically motivated reinforcement learning implementation where the motivation signal is used in addition to the reinforcement learning reward. One can then quickly measure the two agents' performance by adding or removing the motivation signal. Stout and Barto [90] introduced competence progress as a measure to determine which skill the agent can learn at a particular point in time. Low competence progress signal is generated for the skills that are either learned or are too difficult to learn.

In general, it appears that the performance metrics seem to fall into one of the following categories:

- Metrics that compare the performance of an intrinsically motivated reinforcement learning agent with a reinforcement learning agent: This determines how intrinsic motivation improves the performance of reinforcement learning.
- Metrics that characterise the performance of a chosen motivation function: This determines how good or bad the motivation function is.
- Metrics that characterise an emergent behaviour: This shows how intrinsic motivation influences the behaviour of an agent.

Table A.2: Classification of the performance measures seen in the literature on motivated reinforcement learning agent architectures.

Performance Measure	**Publications That Use These Metrics**
Metrics comparing the performance of an intrinsically motivated reinforcement learning agent with a simple reinforcement learning agent	[45] [188] [184] [171] [197] [193] [96] [195] [97]
Metrics for characterising the performance of a chosen motivation function	[190] [98] [184] [192] [189] [87] [183] [191] [194]
Metrics for characterising an emergent behaviour	[90] [169] [171] [172] [173] [174] [176] [177] [180] [181] [182] [198]
Other metrics	[199] [170] [93] [175] [185] [178] [186] [187] [200] [201]

Table A.2 classifies the literature according to the category of performance measure used. This classification shows that several techniques have been proposed for evaluating intrinsically motivated reinforcement learning. However, as with many other aspects of intrinsically motivated reinforcement learning, there is not yet an agreement on which of these techniques should be used as a benchmark for comparing the different architectures.

APPENDIX B: COMPARISON OF REINFORCEMENT LEARNING ALGORITHMS FOR Q-TABLE BASED APPROACHES

This appendix section does a side by side comparison of the Q-Learning [39] and Dyna-Q [40] reinforcement learning algorithms. Both the algorithms are detailed in Chapter 2. Q-Learning is the most commonly used algorithm. It is widely used because it has the least number of parameters to tune and its simplicity. Dyna-Q combines Q-Learning with model learning, i.e. has an internal representation of the transition model. The Q-values are improved using the actual experiences by interacting with its environment and imaginary experiences generated by the transition model. That reduces the actual interactions with the environment making it more performant, especially for robotics applications where the actual interaction can be expensive. These experiments do not make an exhaustive comparison (that already exists in the literature [40]), but the aim is to compare the performance of these algorithms for different task types and help put the results in perspective. The algorithms were tested on the standard benchmark cart-pole problem, a maintenance task, and a maze navigation problem, an achievement task. These task types are defined in detail in Chapter 4.

Problem Definition

Shown in Figure B.1 is a cart-pole problem. A cart carrying a hinged pole is placed on a finite track, and the reinforcement learning agent aims to keep the pole balanced for as long as possible. It is primarily a control problem and can be represented in both continuous and discrete state and action space. For the experiment in this section, discrete state and action space are used. The state space comprises the position of the cart, velocity of the cart, angle of the pole with respect to the cart and angular velocity of the pole. The action space comprises pushing the cart to the left and pushing the cart to the right.

Shown in Figure B.2 is the maze problem. In this contrived problem, the maze consists of a start state and an end state with random walls throughout the maze. The reinforcement learning agent aims to find its way from the start state to the end state. The state space comprises the x and y position numbers of the square blocks of the maze. The action space comprises moving left, right, up and down.

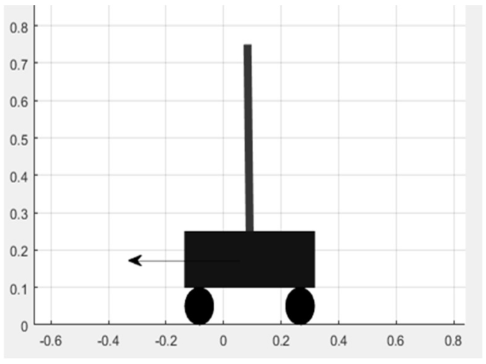


Figure B.1: Cart-pole problem. A cart that carries a hinged pole is placed on a finite track. The aim of the agent is to learn to keep the pole balanced for as long as possible.

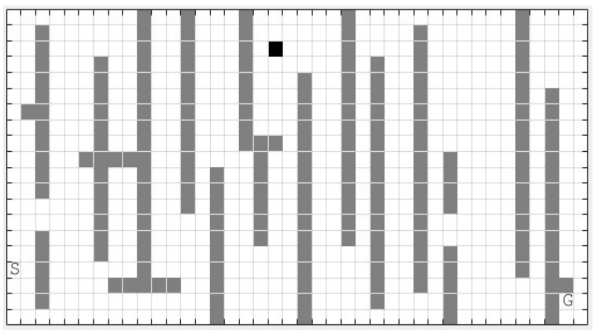
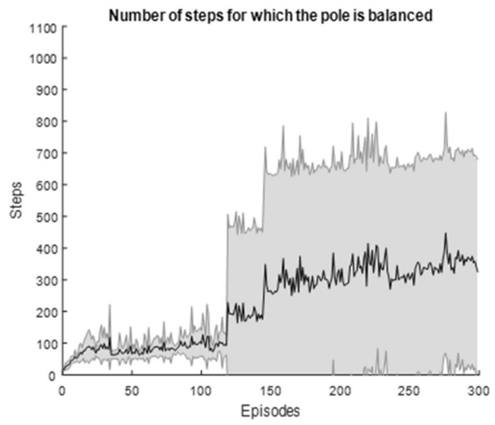


Figure B.2: Maze problem. The square marked 'S' is the start state, and the one marked 'G' is the end state. Grey squares are the walls, and the black square is the agent. The aim of the agent is to find its way through the maze from the square marked as 'S' (bottom left) to the square marked 'G' (bottom right).

Results of the experiment with the cart-pole problem

For both algorithms, the reward for the problem was a small positive value of 1 for every step the pole is kept balanced and a relatively large negative value of -10 when the pole falls. The episode ended when the pole fell or 1000 steps. The experiment was run 10 times for a trial of 300 episodes each. The epsilon-greedy action selection strategy was used with the epsilon parameter set to 0.1 with no decay.



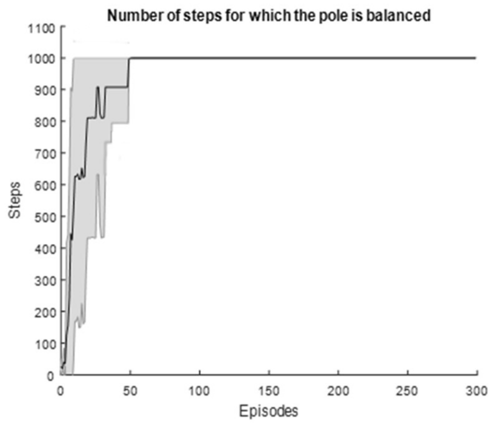


Figure B.3: Q-Learning results for the cart-pole problem. It shows the number of steps for which the pole was balanced. A higher number of steps indicates good performance.

Figure B.4: Dyna-Q results for the cart-pole problem. It shows the number of steps for which the pole was balanced. A higher number of steps indicates good performance.

The plots in Figure B.3 and Figure B.4 show the results for the two algorithms during the training phase. They show the number of steps for which the pole was balanced. The results are for 10 trials, with the black line showing the average number of steps for which the pole was balanced and the shaded region showing the standard deviation.

Results of the experiment with the maze problem

For both algorithms, the reward for the problem was a small negative value of -1 for every step the agent takes and a relatively large positive value of 10 when the agent reaches the goal state. The episode ended when the agent reached the goal state or after 5000 steps. The experiment was run 10 times for a trial of 300 episodes each. The epsilon-greedy action selection strategy was used with the epsilon parameter set to 0.1 with no decay.

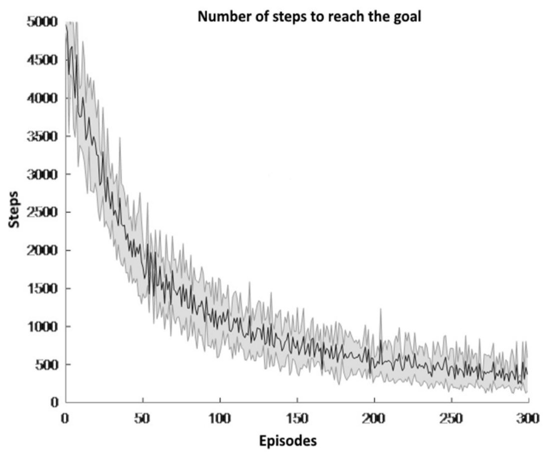


Figure B.5: Q-Learning results for the maze problem. It shows the number of steps the agent takes to reach the goal. A lower number of steps indicates good performance.

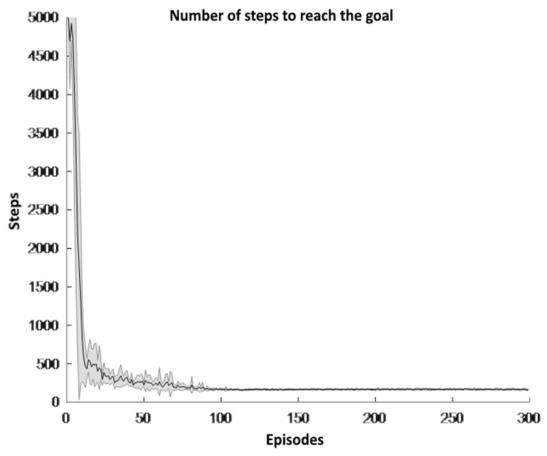


Figure B.6: Dyna-Q results for the maze problem. It shows the number of steps the agent takes to reach the goal. A lower number of steps indicates good performance.

The plots in Figure B.5 and Figure B.6 show the results for the two algorithms during the training phase. They show the number of steps taken by the agent to reach the goal state. The results are for 10 trials, with the black line showing the average number of steps taken by the agent to reach the goal state and the shaded region showing the standard deviation.

Discussion and comparison of the algorithms

Results for the cart-pole problem show that, on average, the agent using Dyna-Q can balance the pole for more steps than the agent using Q-Learning, and the results for the maze problem show that, on average, the agent using Dyna-Q can reach the goal state in fewer steps than the agent using Q-Learning. Also, in both cases, the agent's performance using Dyna-Q is more stable than the agent using Q-Learning, i.e. Dyna-Q converges, and the agent reaches the optimal solution in fewer episodes. This comparison with maintenance and achievement task types helps put the performance of the algorithms in perspective. The table below lists the pros and cons of each of the algorithms.

Table B.1: Comparison of Q-Learning and Dyna-Q algorithms.

Algorithm	**Pros**	**Cons**
Q-Learning	1. Simple implementation, no state transition model required.	1. Slower convergence compared to Dyna-Q 2. Less stable learning compared to Dyna-Q
Dyna-Q	1. Faster convergence compared to Q-Learning. 2. Stable learning compared to Q-Learning. 3. Since there is an internal state transition model, it requires fewer interactions with the real world and is suitable for robotics applications where interaction with the real world can be expensive.	1. Slightly complex implementation compared to Q-Learning. Implementation requires a loop for interaction with the real world and a loop for iteration within the internal state transition model.

Considering the benefits of the Dyna-Q algorithm, it was selected as an algorithm for the experiments in Chapters 3, 4 and 5.

REFERENCES

[1] J. Weng *et al.*, “Autonomous mental development by robots and animals,” *Science (80-.).*, vol. 291, no. 5504, pp. 599–600, 2001.

[2] M. Wooldridge and N. R. Jennings, “Intelligent Agents: Theory and Practice,” *Knowledge Engineering Review*, vol. 10, no. 2. pp. 115–52, 1995.

[3] I. Kotseruba and J. K. Tsotsos, “A Review of 40 Years of Cognitive Architectures research: Core Cognitive Abilities and Practical Applications,” *Artificial Intelligence Review*. 2018.

[4] G. Baldassarre, “What are intrinsic motivations? A biological perspective,” *2011 IEEE International Conference on Development and Learning, ICDL 2011*. Ieee, pp. 1–8, Aug. 2011.

[5] P. Morignot and B. Hayes-Roth, “Why does an Agent Act : Adaptable Motivations for Goal Selection and Generation,” *AAAI Spring Symposium on Representing Mental States and Mechanisms*, no. 1. pp. 97–101, 1995.

[6] S. Franklin and A. Graesser, “Is It an Agent, or Just a Program?: A Taxonomy of Autonomous Agents,” *International Workshop on Agent Theories, Architectures, and Languages*. pp. 21–35, 1996.

[7] L. P. Kaelbling and S. J. Rosenschein, “Action and planning in embedded agents,” *Robotics and Autonomous Systems*, vol. 6, no. 1–2. pp. 35–48, 1990.

[8] J. A. Starzyk, J. T. Graham, and L. Puzio, “Needs, Pains, and Motivations in Autonomous Agents,” *IEEE Transactions on Neural Networks and Learning Systems*. 2016.

[9] W. Duch, R. J. Oentaryo, and M. Pasquier, “Cognitive Architectures: Where do we go from here?,” *Proceedings of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, vol. 171, no. OCTOBER. pp. 122–136, 2008.

[10] K. On Chin, K. S. Gan, R. Alfred, P. Anthony, and D. Lukose, “Agent Architecture: An Overview,” *Transactions on Science and Technology*, vol. 1, no. 1. pp. 18–35, 2014.

[11] K. E. Merrick, “Intrinsic motivation and introspection in reinforcement learning,” *IEEE Transactions on Autonomous Mental Development*, vol. 4. pp. 315–329, 2012.

[12] V. G. Santucci, G. Baldassarre, and M. Mirolli, “GRAIL: A goal-discovering robotic architecture for intrinsically-motivated learning,” *IEEE Transactions on Cognitive and Developmental Systems*, vol. 8, no. 3. pp. 214–231, 2016.

[13] G. Baldassarre, V. G. Santucci, E. Cartoni, and D. Caligiore, “The architecture challenge: Future artificial-intelligence systems will require sophisticated architectures, and knowledge of the brain

might guide their construction," *The Behavioral and brain sciences*, vol. 40. p. e254, 2017.

[14] S. B. Thrun and T. M. Mitchell, "Lifelong Robot Learning," *Robotics and Autonomous Systems*, vol. 15, no. March 1993. pp. 25–46, 1995.

[15] Z. Chen and B. Liu, *Lifelong Machine Learning*, vol. 10, no. 3. 2016.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a tutorial," *Journal of Artificial Intelligence Research*. 1996.

[17] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum Learning," *International Conference on Machine Learning*. 2009.

[18] P. Morignot and B. Hayes-Roth, "Motivated agents," *Knowledge Systems Laboratory Report KSL-96*, vol. 22, no. July. 1996.

[19] F. Dignum and R. Conte, "Intentional Agents and Goal Formation," *Agent Theories, Architectures, and Languages*. pp. 231–243, 1997.

[20] A. Hsissi, H. Allali, and A. Hajami, "Metacognitive Scaffolding Agent Based on BDI Model for Interactive Learning Environments," *International Journal of Computer and Communication Engineering*, vol. 3, no. 2. pp. 97–100, 2014.

[21] C. Colas, P. Fournier, O. Sigaud, M. Chetouani, and P.-Y. Oudeyer, "CURIOUS: Intrinsically motivated modular multi-goal reinforcement learning," *36th Int. Conf. Mach. Learn. ICML 2019*, vol. 2019-June, pp. 2372–2387, 2019.

[22] F. Kaplan and P.-Y. Oudeyer, "Intrinsically Motivated Machines," in *50 Years of Artificial Intelligence*, vol. 71, no. 4–6, Springer, Berlin, Heidelberg, 2007, pp. 303–314.

[23] S. B. Thrun and L. Pratt, *Learning to Learn*. 1998.

[24] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, vol. 4, no. 463–502. pp. 463–502, 1969.

[25] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113. pp. 54–71, 2019.

[26] J. Schmidhuber, "On Learning how to Learn Learning Strategies," *Technical Report FKI-198-94*, vol. 94. pp. 1–20, 1995.

[27] J. T. Graham and J. A. Starzyk, "Transitioning from motivated to cognitive agent model," *Proceedings of the 2013 IEEE Symposium on Computational Intelligence for Human-Like Intelligence, CIHLI 2013*. pp. 9–16, 2013.

[28] C. Castelfranchi and R. Falcone, "Agent Autonomy," no. January 2003. pp. VI, 288, 2003.

[29] F. Tanaka and M. Yamamura, "An approach to lifelong reinforcement learning through multiple

environments," *Proc. of the 6th European Workshop on Learning Robot (EWLR-6)*. pp. 93–99, 1997.

[30] R. S. Sutton, J. Modayil, M. D. T. Degris, P. M. Pilarski, A. White, and D. Precup, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," *10th International Conference on Autonomous Agents and Multiagent Systems 2011, AAMAS 2011*, vol. 2, no. 1972. pp. 713–720, 2011.

[31] D. Isele, "Lifelong Reinforcement Learning On Mobile Robots," University of Pennsylvania, 2018.

[32] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, "Goal representation for BDI agent systems," *Second International Workshop on Programming Multiagent Systems: Languages and Tools*. pp. 9–20, 2005.

[33] B. Price and C. Boutilier, "Accelerating reinforcement learning through imitation," *Journal of Artificial Intelligence Research*, vol. 19. pp. 569–629, 2003.

[34] S. B. Thrun, "A Lifelong Learning Perspective for Mobile Robot Control," *Intelligent Robots and Systems*. pp. 201–214, 1995.

[35] A. Wilson, A. Fern, S. Ray, and P. Tadepalli, "Multi-task reinforcement learning: a hierarchical Bayesian approach," *Proceedings of the 24th international conference on Machine learning*. pp. 1015-1022), 2007.

[36] B. T. Polyak and A. B. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM Journal on Control and Optimization*, vol. 30, no. 4. pp. 838–855, 1992.

[37] D. M. J. Tax, M. Van Breukelen, R. P. W. Duin, and Josef Kittler, "Combining multiple classifiers by averaging or by multiplying?," *Pattern Recognition*, vol. 33, no. 9. pp. 1475–1485, 2000.

[38] K. E. Merrick, N. Siddique, and I. Rano, "Experience-Based Generation of Maintenance and Achievement Goals on a Mobile Robot," *Paladyn, Journal of Behavioral Robotics*. pp. 67–84, 2016.

[39] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q-Learning," *Machine Learning*, vol. 8, no. 3. pp. 279–292, 1992.

[40] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press Cambridge, 1998.

[41] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *33rd International Conference on Machine Learning, ICML 2016*, 2016, vol. 4, pp. 2850–2869.

[42] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1. pp. 181–211, 1999.

[43] D. Precup, R. S. Sutton, and S. Singh, "Theoretical results on reinforcement learning with temporally abstract options," *Machine Learning: European Conference on Machine Learning-98. Springer Berlin Heidelberg*, no. April. pp. 382–393, 1998.

[44] K. E. Merrick and M. Lou Maher, *Motivated reinforcement learning: Curious characters for multiuser games*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[45] S. Singh, A. G. Barto, and N. Chentanez, "Intrinsically motivated reinforcement learning," *18th Annual Conference on Neural Information Processing Systems (NIPS)*, vol. 17. pp. 1281–1288, 2005.

[46] G. A. Carpenter and S. Grossberg, "Adaptive Resonance Theory," no. 617, pp. 1–12, 2002.

[47] A. van Lamsweerde, "Goal-oriented requirements engineering: a guided tour," *Proceedings Fifth IEEE International Symposium on Requirements Engineering*. Toronto, pp. 249–262, 2001.

[48] M. B. van Riemsdijk, M. Dastani, and M. Winikoff, "Goals in agent systems: A unifying framework," *Proc. of 7th International Conference on Autonomous Agents & Multiagent Systems*. pp. 713–720, 2008.

[49] K. E. Merrick, "Modelling Motivation For Experience-Based Attention Focus In Reinforcement Learning," School of Information Technologies, University of Sydney, 2007.

[50] D. L. Silver, Q. Yang, and L. Li, "Lifelong Machine Learning Systems : Beyond Learning Algorithms," *AAAI Spring Symposium Series*, no. Solomonoff 1989. pp. 49–55, 2013.

[51] V. G. Santucci, G. Baldassarre, and M. Mirolli, "Biological cumulative learning through intrinsic motivations: a simulated robotic study on the development of visually-guided reaching," *Proceedings of the Tenth International Conference on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, vol. 0. pp. 121–128, 2010.

[52] A. Bonarini, A. Lazaric, and M. Restelli, "Incremental Skill Acquisition for Self-motivated Learning Animats," *Proceedings of the Ninth International Conference on Simulation of Adaptive Behavior (SAB-06)*, vol. 4095. pp. 357–368, 2006.

[53] A. Baranes and P.-Y. Oudeyer, "Intrinsically motivated goal exploration for active motor learning in robots: A case study," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*. pp. 1766–1773, 2010.

[54] A. Baranes and P.-Y. Oudeyer, "Maturationally-constrained competence-based intrinsically motivated learning," *2010 IEEE 9th International Conference on Development and Learning*. Ieee, pp. 197–203, Aug. 2010.

[55] P. Maes, "Modeling Adaptive Autonomous Agents," *Artificial Life*, vol. 1, no. 1_2. pp. 135–162, 1993.

[56] K. E. Merrick, "Value Systems for Developmental Cognitive Robotics : A Survey," *Cognitive Systems Research*, vol. 41, no. August. Elsevier B.V., pp. 38–55, 2016.

[57] S. Doncieux *et al.*, "Open-ended learning: A conceptual framework based on representational redescription," *Frontiers in Neurorobotics*, vol. 12, no. SEP. pp. 1–6, 2018.

[58] E. Cartoni, D. Montella, J. Triesch, and G. Baldassarre, “An open-ended learning architecture to face the REAL 2020 simulated robot competition.” pp. 1–21, 2020.

[59] A. W. Moore, L. C. Baird, and L. P. Kaelbling, “Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs,” *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2. pp. 1318–1321, 1999.

[60] V. G. Santucci, E. Cartoni, B. C. da Silva, and G. Baldassarre, “Autonomous Open-Ended Learning of Interdependent Tasks,” *arXiv:1905.02690*. pp. 0–5, 2019.

[61] V. G. Santucci, D. Montella, B. C. da Silva, and G. Baldassarre, “Autonomous learning of multiple, context-dependent tasks,” *arXiv:2011.13847*, pp. 1–15, 2020.

[62] U. Jaidee, H. Muñoz-Avila, and D. W. Aha, “Integrated learning for goal-driven autonomy,” *IJCAI International Joint Conference on Artificial Intelligence*. pp. 2450–2455, 2011.

[63] M. Hanheide *et al.*, “A Framework for Goal Generation and Management,” *Proceedings of the AAAI Workshop on Goal-Directed Autonomy*. 2010.

[64] J. E. Laird, “Toward cognitive robotics,” *Unmanned Systems Technology XI*, vol. 7332, no. May 2009. p. 73320Z, 2009.

[65] S. Doncieux *et al.*, “DREAM Architecture: a Developmental Approach to Open-Ended Learning in Robotics,” *arXiv:2005.062231*, pp. 1–29, 2020.

[66] A. McGovern and A. G. Barto, “Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density,” *Proceedings of the 18th International Conference on Machine Learning*. pp. 361–368, 2001.

[67] E. Brunskill and L. Li, “PAC-inspired Option Discovery in Lifelong Reinforcement Learning,” *Proceedings of the 31st International Conference on Machine Learning*, vol. 32. pp. 316–324, 2014.

[68] M. Gösgens, A. Tikhonov, and L. Prokhorenkova, “Systematic Analysis of Cluster Similarity Indices: How to Validate Validation Measures,” *International Conference on Machine Learning*. pp. 3799–3808, 2021.

[69] S. Marsland, U. Nehmzow, and J. Shapiro, “On-line novelty detection for autonomous mobile robots,” *Robotics and Autonomous Systems*, vol. 51, no. 2–3. pp. 191–206, 2005.

[70] R. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE journal on robotics and automation*, vol. 2, no. 1. pp. 14–23, 1986.

[71] A. S. Rao and M. P. Georgeff, “BDI agents: From theory to practice.,” *Proceedings of the First International Conference on Multiagent Systems*, vol. 95. pp. 312–319, 1995.

[72] G. Regev and A. Wegmann, “Where do Goals Come from : The Underlying Principles of Goal-Oriented Requirements Engineering,” *13th IEEE International Conference on Requirements*

Engineering. pp. 353–362, 2005.

[73] D. Dewey, “Reinforcement Learning and the Reward Engineering Principle,” *AAAI Spring Symposium Series*. pp. 1–8, 2014.

[74] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, “Concrete Problems in AI Safety,” *arXiv:1606.06565*. pp. 1–29, 2016.

[75] M. Andrychowicz *et al.*, “Hindsight Experience Replay,” in *Advances in Neural Information Processing Systems*, 2017, vol. December, pp. 5049–5059.

[76] J. Kober and J. Peters, “Imitation and reinforcement learning,” *IEEE Robotics and Automation Magazine*, vol. 17, no. 2. pp. 55–62, 2010.

[77] P. Ranchod, B. Rosman, and G. D. Konidaris, “Nonparametric Bayesian reward segmentation for skill discovery using inverse reinforcement learning,” *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. pp. 471–477, 2015.

[78] H. B. Suay, T. Brys, M. E. Taylor, and S. Chernova, “Learning from Demonstration for Shaping through Inverse Reinforcement Learning,” *International Conference on Autonomous Agents & MultiAgent Systems*. pp. 429–437, 2016.

[79] J. Schmidhuber, “What’s Interesting,” *In Abstract Collection of SNOWBIRD: Machines That Learn*. pp. 1–23, 1997.

[80] J. Schmidhuber, “Artificial curiosity based on discovering novel algorithmic predictability through coevolution,” *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, vol. 3. pp. 1612–1618, 1999.

[81] R. M. Ryan and E. L. Deci, “Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions,” *Contemporary educational psychology*, vol. 25, no. 1. pp. 54–67, 2000.

[82] A. Aubret, L. Matignon, and S. Hassas, “A survey on intrinsic motivation in reinforcement learning.” arXiv preprint arXiv:1908.06976, 2019.

[83] G. Baldassarre and M. Mirolli, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Berlin, Heidelberg: Springer Heidelberg, 2013.

[84] V. G. Santucci, P. Y. Oudeyer, A. Barto, and G. Baldassarre, “Editorial: Intrinsically motivated open-ended learning in autonomous robots,” *Frontiers in Neurorobotics*, vol. 13, no. January. 2020.

[85] P.-Y. Oudeyer and F. Kaplan, “What is Intrinsic Motivation? A Typology of Computational Approaches,” *Frontiers in neurorobotics*, vol. 1, no. November. p. 6, Jan. 2007.

[86] G. Baldassarre, “Intrinsic Motivations and Open-ended Learning,” *arXiv preprint arXiv:1912.13263*. 2019.

[87] J. Achiam and S. Sastry, “Surprise-based intrinsic motivation for deep reinforcement learning,” *arXiv Prepr. arXiv1703.01732*, pp. 1–14, 2017.

[88] D. E. Berlyne, “Novelty, complexity, and hedonic value,” *Perception & Psychophysics*, vol. 8, no. 5. pp. 279–286, 1970.

[89] M. Mirolli and G. Baldassarre, “Functions and Mechanisms of Intrinsic Motivations. The Knowledge Versus Competence Distinction,” in *Intrinsically Motivated Learning in Natural and Artificial Systems*, 2013, pp. 49–72.

[90] A. Stout and A. G. Barto, “Competence progress intrinsic motivation,” *2010 IEEE 9th International Conference on Development and Learning, ICDL-2010 - Conference Program*. Ieee, pp. 257–262, Aug. 2010.

[91] A. Bonarini, A. Lazaric, and M. Restelli, “Learning Reusable Skills through Self-Motivation,” *Proceedings of the ICML Workshop on Structural Knowledge Transfer for Machine Learning*. 2006.

[92] P. Sequeira, F. S. Melo, and A. Paiva, “Emotion-Based Intrinsic Motivation for Reinforcement Learning Agents,” *Affective Computing and Intelligent Interaction*, vol. 6974. pp. 326–336, 2011.

[93] E. Temel, B. J. Grzyb, and S. Sariel, “Learning graspability of unknown objects via intrinsic motivation,” *CEUR Workshop Proceedings*, vol. 1315. pp. 98–109, 2014.

[94] B. J. Grzyb, J. Boedecker, M. Asada, A. P. del Pobil, and L. B. Smith, “Between Frustration and Elation: Sense of Control Regulates the lntrinsic Motivation for Motor Learning,” *Lifelong Learning, Advancement of Artificial Intelligence Workshop*. pp. 10–15, 2011.

[95] Q. Ma, G. Pei, and L. Meng, “Inverted U-shaped curvilinear relationship between challenge and one’s intrinsic motivation: Evidence from event-related potentials,” *Frontiers in Neuroscience*, no. 11. p. 131, 2017.

[96] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, “Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation,” *Computing Research Repository*. pp. 1–13, 2016.

[97] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, “Unifying count-based exploration and intrinsic motivation,” in *Advances in Neural Information Processing Systems*, 2016, pp. 1479–1487.

[98] V. G. Santucci, G. Baldassarre, and M. Mirolli, “Which is the best intrinsic motivation signal for learning multiple skills?,” *Frontiers in neurorobotics*, vol. 7. 2013.

[99] A. Laud and G. DeJong, “Reinforcement Learning and Shaping: Encouraging Intended Behaviors.,” *Proceedings of International Conference on Machine Learning*. 2002.

[100] A. C. Tenorio-Gonzalez, E. F. Morales, and L. Villaseñor-Pineda, “Dynamic reward shaping:

Training a robot by voice," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6433 LNAI, no. 214262. pp. 483–492, 2010.

[101] B. Marthi, "Automatic shaping and decomposition of reward functions," *Proceedings of the 24th international conference on Machine learning - ICML '07*. pp. 601–608, 2007.

[102] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations : Theory and application to reward shaping," *Sixteenth International Conference on Machine Learning*, vol. 3. pp. 278–287, 1999.

[103] G. D. Konidaris and A. G. Barto, "Autonomous Shaping : Knowledge Transfer in Reinforcement Learning," *Proceedings of the 23rd International Conference on Machine Learning*. pp. 489–496, 2006.

[104] M. J. Matarić, "Reinforcement Learning in the Multi-Robot Domain," *Autonomous Robots*, vol. 4, no. 1. pp. 73–83, 1997.

[105] A. Y. Ng, "Shaping and policy search in reinforcement learning," University of California, Berkeley, 2003.

[106] P. Fournier, "Intrinsically Motivated and Interactive Reinforcement Learning : a Developmental Approach," Sorbonne University, 2019.

[107] K. V. Hindriks and M. B. Van Riemsdijk, "Satisfying Maintenance Goals," *In International Workshop on Declarative Agent Languages and Technologies*, vol. 4897 LNAI. pp. 86–103, 2007.

[108] A. J. Elliot, *Handbook of Approach and Avoidance Motivation*. 2008.

[109] S. Duff, J. Harland, and J. Thangarajah, "On Proactivity and Maintenance Goals," *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems - AAMAS '06*. p. 1033, 2006.

[110] A. Baraldi, "Simplified ART: A new class of ART algorithms," *International Computer Science Institute*, no. Technical Report, TR 98-0041998. 1998.

[111] M. Dastani and M. Winikoff, "Rich Goal Types in Agent Programming," in *In The 10th International Conference on Autonomous Agents and Multiagent Systems*, 2011, pp. 405–412.

[112] Ö. Simsek and A. G. Barto, "Skill characterization based on betweenness," *Advances in Neural Information Processing Systems 21*. pp. 1497–1504, 2009.

[113] Ö. Simsek, A. P. Wolfe, and A. G. Barto, "Identifying useful subgoals in reinforcement learning by local graph partitioning," *In Proceedings of the Twenty Second International Conference on Machine Learning (ICML 2005)*. pp. 816–823, 2005.

[114] Ö. Şimşek and A. G. Barto, "Using relative novelty to identify useful temporal abstractions in

reinforcement learning," *Twenty-first international conference on Machine learning (ICML '04)*. pp. 751–758, 2004.

[115] G. D. Konidaris and A. G. Barto, "Skill discovery in continuous reinforcement learning domains using skill chaining," *Advances in Neural Information Processing Systems*. pp. 1–9, 2009.

[116] S. Forestier, Y. Mollard, and P.-Y. Oudeyer, "Intrinsically Motivated Goal Exploration Processes with Automatic Curriculum Learning," *arXiv preprint arXiv:1708.02190*. pp. 1–21, 2017.

[117] V. Sperati and G. Baldassarre, "Bio-Inspired Model Learning Visual Goals and Attention Skills Through Contingencies and Intrinsic Motivations," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 10, no. 2. pp. 326–344, 2018.

[118] N. Siddique, P. Dhakan, I. Rano, and K. E. Merrick, "A review of the relationship between novelty, intrinsic motivation and reinforcement learning," *Paladyn, Journal of Behavioral Robotics*, vol. 8, no. 1. pp. 58–69, 2017.

[119] P.-Y. Oudeyer, "Intelligent Adaptive Curiosity: a source of Self-Development," *Science*, vol. 117. pp. 127–130, 2004.

[120] M. Rolf, J. J. Steil, and M. Gienger, "Bootstrapping inverse Kinematics with Goal Babbling," *2010 IEEE 9th International Conference on Development and Learning, ICDL-2010 - Conference Program*, no. May 2014. pp. 147–154, 2010.

[121] V. G. Santucci, G. Baldassarre, and M. Mirolli, "Intrinsic motivation mechanisms for competence acquisition," *IEEE International Conference on Development and Learning*. pp. 1–6, 2012.

[122] B. Bakker and J. Schmidhuber, "Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization," *Proc. of the 8th Conference on Intelligent Autonomous Systems*. 2004.

[123] D. Held, X. Geng, C. Florensa, and P. Abbccl, "Automatic Goal Generation for Reinforcement Learning Agents," *35th International Conference on Machine Learning, ICML 2018*, vol. 4. pp. 2458–2471, 2018.

[124] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*, 2nd ed. Springer series in statistics, 2001.

[125] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, "Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions. arXiv:1901.01753v3." pp. 1–28, 2019.

[126] O.-E. L. Team *et al.*, "Open-Ended Learning Leads to Generally Capable Agents." 2021, [Online]. Available: http://arxiv.org/abs/2107.12808.

[127] B. van Niekerk, S. James, A. Earle, and B. Rosman, "Composing Value Functions in Reinforcement Learning," *International Conference on Machine Learning*. 2019.

[128] A. Lazaric, “Transfer in Reinforcement Learning : a Framework and a Survey,” in *Reinforcement Learning - State of the art*, vol. 12, Springer, 2012, pp. 143–173.

[129] M. E. Taylor and P. Stone, “Transfer Learning for Reinforcement Learning Domains : A Survey,” *Journal of Machine Learning Research*, vol. 10. pp. 1633–1685, 2009.

[130] X. Bin Peng, M. B. Chang, G. Zhang, P. Abbeel, and S. Levine, “MCP: Learning Composable Hierarchical Control with Multiplicative Compositional Policies,” *Advances in Neural Information Processing Systems*. pp. 3686–3697, 2019.

[131] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor, “A Deep Hierarchical Approach to Lifelong Learning in Minecraft,” *31st AAAI Conference on Artificial Intelligence, AAAI 2017*. pp. 1553–1561, 2017.

[132] B. Wu, J. K. Gupta, and M. J. Kochenderfer, “Model Primitive Hierarchical Lifelong Reinforcement Learning,” *18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019)*. Montreal, Canada, 2019.

[133] G. Auda and M. Kamel, “Modular neural networks: a survey,” *International journal of neural systems*, vol. 9, no. 2. pp. 129–151, 1999.

[134] S. Singh, “Transfer of Learning by Composing Solutions of Elemental Sequential Tasks,” *Machine Learning*. pp. 323–339, 1992.

[135] Y. Lee, S. Sun, S. Somasundaram, E. Hu, and J. J. Lim, “Composing Complex Skills by Learning Transition Policies,” *International Conference in Learning Representations*. pp. 1–19, 2019.

[136] A. H. Qureshi, J. J. Johnson, Y. Qin, B. Boots, and M. C. Yip, “Composing Ensembles of Policies with Deep Reinforcement Learning,” *International Conference in Learning Representations*. pp. 1–16, 2020.

[137] S. B. Thrun, *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, 1996.

[138] C. Drummond, “Accelerating Reinforcement Learning by Composing Solutions of Automatically Identified Subtasks,” *Journal of Artificial Intelligence Research*, vol. 16. pp. 59–104, 2002.

[139] M. A. Wiering, W. Maikel, and D. Madalina, “Model-Based Multi-Objective Reinforcement Learning,” in *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2014, vol. 30, no. 6.

[140] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier, “Hierarchical Solution of Markov Decision Process using Macro-actions,” in *Fourteenth conference on Uncertainty in artificial intelligence*, 1998, pp. 220–229.

[141] M. Stolle, “Automated discovery of options in reinforcement learning,” McGill University, 2004.

[142] P. L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," *31st AAAI Conference on Artificial Intelligence, AAAI 2017*. pp. 1726–1734, 2017.

[143] T. Dietterich, "An Overview of MaxQ Hierarchical Reinforcement Learning," *International Symposium on Abstraction, Reformulation, and Approximation*. pp. 26–44, 2000.

[144] Z. Kalmár, C. Szepesvári, and A. Lorincz, "Module-Based Reinforcement Learning: Experiments with a Real Robot," *Autonomous Robots*, vol. 5, no. 3–4. pp. 273–295, 1998.

[145] E. Uchibe, M. Asada, and K. Hosoda, "Behavior coordination for a mobile robot using modular reinforcement learning," *IEEE International Conference on Intelligent Robots and Systems*, vol. 3. pp. 1329–1336, 1996.

[146] H. Sahni, S. Kumar, F. Tejani, and C. L. Isbell, "Learning to compose skills," 2017.

[147] C. Devin, A. Gupta, T. Darrell, P. Abbeel, and S. Levine, "Learning Modular Neural Network Policies for Multi-Task and Multi-Robot Transfer," *IEEE International Conference on Robotics and Automation (ICRA)*. 2017.

[148] C. Drummond, "Composing functions to speed up reinforcement learning in a changing world," *European Conference on Machine Learning*. pp. 370–381, 1998.

[149] È. Pairet, P. Ardón, M. Mistry, and Y. Petillot, "Learning and Composing Primitive Skills for Dual-Arm Manipulation," *20th Annual Conference of Towards Autonomous Robotic Systems*, vol. 11649 LNAI. pp. 65–77, 2019.

[150] T. Haarnoja, V. Pong, A. Zhou, M. Dalal, P. Abbeel, and S. Levine, "Composable Deep Reinforcement Learning for Robotic Manipulation," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, no. 1. pp. 6244–6251, 2018.

[151] C. Simpkins and C. L. Isbell, "Composable Modular Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33. pp. 4975–4982, 2019.

[152] G. Berseth, C. Xie, P. Cernek, and M. Van de Panne, "Progressive Reinforcement Learning with Distillation for Multi-Skilled Motion Control," *International Conference on Learning Representations*. pp. 1–15, 2018.

[153] W. M. Czarnecki *et al.*, "Mix & match - Agent curricula for reinforcement learning," *35th International Conference on Machine Learning, ICML 2018*, vol. 3. pp. 1761–1773, 2018.

[154] E. Todorov, "Compositionality of Optimal Control Laws," *Advances in Neural Information Processing Systems, 2009*, vol. 3. pp. 1856–1864, 2009.

[155] P. Dhakan, K. E. Merrick, I. Rano, and N. Siddique, "Intrinsic rewards for maintenance, approach, avoidance, and achievement goal types," *Frontiers in Neurorobotics*, vol. 12, no. October. 2018.

[156] M. P. Deisenroth, "Efficient Reinforcement Learning using Gaussian Processes," Karlsruhe Institute

of Technology, 2010.

[157] M. Asada, E. Uchibe, S. Noda, S. Tawaratsumida, and K. Hosoda, “Coordination of multiple behaviors acquired by a vision-based reinforcement learning,” *IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, vol. 2. pp. 917–924, 1994.

[158] S. Russell and A. L. Zimdars, “Q-Decomposition for Reinforcement Learning Agents,” *20th International Conference on Machine Learning*. pp. 656–663, 2003.

[159] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*. pp. 529–533, 2015.

[160] S. E. Yuksel, J. N. Wilson, and P. D. Gader, “Twenty years of mixture of experts,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 8. pp. 1177–1193, 2012.

[161] S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *The Annals of Mathematical Statistics*. 1951.

[162] B. De Alwis, G. C. Murphy, and S. Minto, “Creating a cognitive metric of programming task difficulty,” *Proceedings - International Conference on Software Engineering*, no. November. pp. 29–32, 2008.

[163] O. Macindoe, M. Lou Maher, and K. E. Merrick, “Agent Based Intrinsically Motivated Intelligent Environments,” *Handbook on Mobile and Ubiquitous Computing: Innovations and Perspectives*. 2007.

[164] J. Kober and J. Peters, “Policy search for motor primitives in robotics,” in *Advances in Neural Information Processing Systems*, 2008, vol. 84, no. 1–2, pp. 171–203.

[165] P. Ruvolo and E. Eaton, “ELLA: An efficient lifelong learning algorithm,” *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, no. 1. pp. 507–515, 2013.

[166] J. García and F. Fernández, “A comprehensive survey on safe reinforcement learning,” *Journal of Machine Learning Research*, vol. 16. pp. 1437–1480, 2015.

[167] R. M. Ryan and E. L. Deci, “Self-Determination Theory and the Facilitation of Intrinsic Motivation, Social Development, and Well-Being,” vol. 55, no. 1, pp. 68–78, 2000.

[168] A. Mark and D. Timothy, “Challenge Point: A Framework for Conceptualizing the Effects of Various Practice Conditions in Motor Learning,” *Journal of Motor Behavior*, vol. 36, no. 2. pp. 212–224, 2004.

[169] H. Ismail, K. E. Merrick, and M. Barlow, “Self-motivated learning of achievement and maintenance tasks for non-player characters in computer games,” *2014 IEEE Symposium on Computational Intelligence for Human-Like Intelligence, Proceedings*. pp. 0–7, 2015.

[170] L. Meeden, J. B. Marshall, and D. Blank, “Self-motivated, task-independent reinforcement learning

for robots," *AAAI Fall Symposium on RealWorld Reinforcement Learning*. 2004.

[171] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2. pp. 70–82, Jun. 2010.

[172] K. E. Merrick and M. Lou Maher, "Motivated reinforcement learning for adaptive characters in open-ended simulation games," *Proceedings of the international conference on Advances in computer entertainment technology - ACE '07*. ACM Press, New York, New York, USA, p. 127, 2007.

[173] K. E. Merrick and M. Lou Maher, "Motivated reinforcement learning for non-player characters in persistent computer game worlds," *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology - ACE '06*. ACM Press, New York, New York, USA, p. 3, 2006.

[174] H. Ngo, M. Luciw, A. Forster, and J. Schmidhuber, "Learning skills from play: Artificial curiosity on a Katana robot arm," *Proceedings of the International Joint Conference on Neural Networks*. Brisbane, Australia, pp. 1–8, 2012.

[175] L. Pape *et al.*, "Learning tactile skills through curious exploration," *Frontiers in Neurorobotics*, vol. 6, no. July. pp. 1–16, 2012.

[176] M. Frank, J. Leitner, M. Stollenga, A. Förster, and J. Schmidhuber, "Curiosity driven reinforcement learning for motion planning on humanoids.," *Frontiers in neurorobotics*, vol. 7, no. January. p. 25, 2014.

[177] D. Di Nocera, A. Finzi, S. Rossi, and M. Staffa, "The role of intrinsic motivations in attention allocation and shifting," *Frontiers in Psychology*, vol. 5, no. April. pp. 1–15, 2014.

[178] M. Luciw, V. R. Kompella, S. Kazerounian, and J. Schmidhuber, "An intrinsic value system for developing multiple invariant representations with incremental slowness learning," *Frontiers in Neurorobotics*, vol. 7, no. May. pp. 1–19, 2013.

[179] C. Zhang, Y. Zhao, J. Triesch, and B. E. Shi, "Intrinsically Motivated Learning of Visual Motion Perception and Smooth Pursuit," *Proc. of IEEE International Conference on Robotics and Automation*. pp. 1902–1908, 2014.

[180] F. Benureau *et al.*, "Intrinsic Motivations for Forming Actions and Producing Goal Directed Behaviour," in *Deliverable for the IM-CLeVeR Spring School at the Capo Caccia Cognitive Neuromorphic Engineering Workshop*, 2011, pp. 1–9.

[181] K. E. Merrick and T. Scully, "Modelling affordances for the control and evaluation of intrinsically motivated robots," *Proceedings of the 2009 Australasian Conference on Robotics and Automation, ACRA 2009*. 2009.

[182] A. G. Barto, S. Singh, and R. L. Lewis, “Intrinsically Motivated Machines,” *Neurocomputing*, vol. 71, no. 4–6. p. 398, 2008.

[183] E. Uchibe and K. Doya, “Constrained reinforcement learning from intrinsic and extrinsic rewards,” *2007 IEEE 6th International Conference on Development and Learning, ICDL*. pp. 163–168, 2007.

[184] A. G. Barto and Ö. Simsek, “Intrinsic motivation for reinforcement learning systems,” *Proceedings of the Thirteenth Yale Workshop on Adaptive and Learning Systems*. pp. 113–118, 2005.

[185] V. R. Kompella, M. Stollenga, M. Luciw, and J. Schmidhuber, “Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots,” *Artificial Intelligence*. pp. 1–42, 2015.

[186] A. Stout, G. D. Konidaris, and A. G. Barto, “Intrinsically Motivated Reinforcement Learning : A Promising Framework For Developmental Robot Learning,” *In Proceedings of the AAAI Spring Symposium on Developmental Robotics, Stanford University, Stanford, CA*. pp. 1–6, 2005.

[187] C. M. Vigorito and A. G. Barto, “Intrinsically Motivated Hierarchical Skill Learning in Structured Environments,” *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2. pp. 132–143, 2010.

[188] A. G. Barto, S. Singh, and N. Chentanez, “Intrinsically motivated learning of hierarchical collections of skills,” *Proceedings of the Third International Conference on Development and Learning*. pp. 112–119, 2004.

[189] M. Schembri, M. Mirolli, and G. Baldassarre, “Evolving internal reinforcers for an intrinsically motivated reinforcement-learning robot,” *2007 IEEE 6th International Conference on Development and Learning, ICDL*. Ieee, pp. 282–287, Jul. 2007.

[190] P. Raif and J. A. Starzyk, “Motivated learning in autonomous systems,” *Proceedings of the international conference on Neural Networks*, no. Ml. pp. 603–610, 2011.

[191] M. Schembri, M. Mirolli, and G. Baldassarre, “Evolution and learning in an intrinsically motivated reinforcement learning robot,” *Advances in Artificial Life*. pp. 294–303, 2007.

[192] P.-Y. Oudeyer, F. Kaplan, and V. V Hafner, “Intrinsic Motivation Systems for Autonomous Mental Development,” *IEEE Transactions On Evolutionary Computation*, vol. 2, no. 2. pp. 265–286, 2007.

[193] G. Masuyama, A. Yamashita, and H. Asama, “Selective exploration exploiting skills in hierarchical reinforcement learning framework,” *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. pp. 692–697, 2013.

[194] J. H. Metzen and F. Kirchner, “Incremental learning of skill collections based on intrinsic motivation,” *Frontiers in Neurorobotics*, vol. 7, no. July. pp. 1–12, 2013.

[195] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman, “Deep Successor Reinforcement

Learning," *arXiv:1606.02396*. p. 10, 2016.

[196] K. E. Merrick, "A Comparative Study of Value Systems for Self-Motivated Exploration and Learning by Robots," *IEEE Transactions on Autonomous Mental Development*, vol. 2. pp. 1–15, 2010.

[197] V. Soni and S. Singh, "Reinforcement Learning of Hierarchical Skills on the Sony Aibo robot," *5th International Conference on Development and Learning*. 2006.

[198] J. Mugan and B. Kuipers, "Towards the Application of Reinforcement Learning to Undirected Developmental Learning," *Proceedings of the Eighth International Conference on Epigenetic Robotics (EpiRob-08)*. pp. 85–92, 2008.

[199] J. B. Marshall, D. Blank, and L. Meeden, "An emergent framework for self-motivation in developmental robotics," *Proceedings of the Third International Conference on Development and Learning ICDL 2004*. pp. 104–111, 2004.

[200] S. Hart, S. Sen, and R. Grupen, "Intrinsically motivated hierarchical manipulation," *Proceedings - IEEE International Conference on Robotics and Automation*. pp. 3814–3819, 2008.

[201] R. L. Lewis, S. Singh, and A. G. Barto, "Where Do Rewards Come From?," *Proceedings of the International Symposium on AI-Inspired Biology*. pp. 2601–2606, 2010.