VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Computer Science

SYNTHESISING CONTINUOUS CONTROL PROGRAMS FROM A DEEP REINFORCEMENT LEARNING AGENT

Anandu Sreekumar

2025-2026

Promotor: Prof. Dr. Ann Nowé
Advisor: Senne Deproost

Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in Applied Sciences and Engineering: Computer Science

SYNTHESE VAN CONTINUE CONTROLEPROGRAMMA'S VAN EEN DEEP REINFORCEMENT LEARNING AGENT

Anandu Sreekumar

2025-2026

Promotor: Prof. Dr. Ann Nowé
Advisor: Senne Deproost

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

Scarcity of skilled controllers in industrial automation systems. Can we apply Machine Learning (ML) to solve this? In ML, the agents learn from data without the need for explicit programming. This adaptability of ML is the key to solving several problem statements. Among the subdivisions of ML, Reinforcement Learning (RL) is a type of learning from experience and finding an optimal behaviour by interacting with the environment. However, the simple RL cannot solve real-world problems; to account for this, the Deep Reinforcement Learning (DRL) was introduced. DRL comprises using neural networks to find the optimal behaviour. But the DRL is accompanied by the core problem of neural networks, i.e., the black box nature.

To address the black box nature and make the agent explain its behaviour, Explainable Artificial Intelligence (XAI) was introduced. The benefit of XAI is that it makes the agent's decision transparent and understandable for humans. In XAI, another subfield is introduced, the Explainable Reinforcement Learning (XRL). XRL explains an RL agent in multiple ways. Program-based explainability is one of the key ways in XRL, which uses a Domain-Specific Language (DSL) program to explain the agent's behaviour.

This thesis is an extended version of Trivedi et al.'s LEAPS framework, which is a program-based explainability approach in a discrete environment. The LEAPS framework comprises two stages. The first stage is the training of a Variational Autoencoder (VAE) model on a dataset of valid DSL programs with a goal of deconstructing and reconstructing the original program. The second stage comprises an RL loop guided search for parameters in the latent space that produce the best-performing programs to solve a given task. Although the LEAPS framework showed promising results in generating programmatic policies for complex behaviours, it was confined to a discrete environment and did not generalise well in real-world applications, since real-world problem statements are continuous in nature. To address this limitation, we extended the LEAPS framework to work in a continuous environment. For this adaptation, the core structure of the framework is modified, a custom DSL is designed, and a custom continuous environment is created.

We created a custom environment with dynamic goal positions and fixed obstacle for this experiment, and the objective of the agent is to reach the goal without hitting the obstacle. Later, we increased the complexity by making the goal position random for each training episode. Notably, the agent was able to solve this environment in an optimised way.

To validate this experiment, we ran a total of 250 runs and computed the results. The results showed the agent's consistent behaviour in reaching the goal in an optimised way, achieving a stable return between 50 and 60. This shows that the framework can be applied to real-world problem statements by tuning the DSL and VAE, thereby providing a potential solution for the scarcity of skilled controllers capable of operating in dynamic and uncertain environments. However, this experiment is designed for a custom environment; for each problem statement and complexity, the design of the DSL and VAE differ, so designing environment-specific DSL and VAE is one of the key limitations in this approach. This experiment focuses on a single objective, but in a real-world problem, it may be multi-objective. This thesis work can be extended to a multi-objective problem with proper design changes.

Acknowledgements

I would like to thank my promotor, Prof. Dr Ann Nowé and advisor Senne Deproost, for providing me the opportunity to work on this thesis topic. I am especially grateful to Prof. Dr Ann Nowé for allowing me to register my thesis; without her help, I wouldn't have gotten this opportunity. I am deeply thankful to Senne Deproost for the efforts, guidance and the positive attitude throughout this journey. His willingness to help and his ability to approach challenges with a smile made a significant difference during the course of this research. There were scenarios where I was doubtful whether I could finish this work within the available time frame, and his support and guidance helped me overcome those moments and reach this stage.

Resuming studies after six years of professional experience was a big step for me. Getting into the rhythm of studies as an international student in one of the challenging specialisations was not an easy path. I'm thankful to my Professors for the proper guidance and for providing the space to ask questions freely and for giving suggestions throughout the journey.

I would like to express my sincere thanks to my family for supporting me, even through situations that were hard to manage. Without their support, I wouldn't have reached this stage. As I firmly believe that for every effort there will be a good part at the end, if it's not the good part, then it's not the end.

I want to thank my friends for supporting and cheering me up when I was having a hard time in my studies and academic work.

Finally, on the research side, I would like to acknowledge Trivedi et al.'s LEAPS framework, which was the foundation for my thesis work. It was a privilege and inspiration to extend this outstanding framework.

Anandu Sreekumar

Acronyms

A3C	Asynchronous Advantage Actor-Critic
AI	Artificial Intelligence
ANN	Artificial Neural Networks
DP	Dynamic Programming
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
IA	Interpretable Agent
IE	Intrinsic Explainability
LEAPS	Learning Embeddings for Latent Program Synthesis
MC	Monte Carlo Methods
MDP	Markov Decision Process
ML	Machine Learning
NDPS	Neurally Directed Program Search
PHE	Post Hoc Explainability
PIRL	Programmatically Interpretable Reinforcement Learning
PPO	Proximal Policy Optimization
PRL	Programmatic Reinforcement Learning
RL	Reinforcement Learning
TD	Temporal Difference
TRPO	Trust Region Policy Optimization
VAE	Variational Autoencoder
XAI	Explainable Artificial Intelligence
XRL	Explainable Reinforcement Learning

Contents

Chapter 1

Introduction

1.1 Context

The scarcity of skilled system controllers in industrial automation is increasing. Machine Learning (ML) has developed as a prominent technology in tackling these problems s [1]. ML is a subfield of Artificial Intelligence (AI) where the agent learns from data (offline or online) without explicitly programming it or without hand-crafted rules [1]. ML can be subdivided into three domains:

- **Supervised Learning** - the model learns a target function from labelled training examples [1].
- **Unsupervised Learning** - the model discovers structures from unlabelled data [1].
- **Reinforcement Learning (RL)** - the model learns optimal behaviour by interacting with the environment [2].

Each of these domains has its own applicability. In the context of system controllers, RL has proven particularly effective. RL is a machine learning paradigm in which an agent learns optimal behaviour through direct interaction with its environment[2]. This paradigm is built upon two distinct bodies of work. The first body of work focused on the psychology of animal learning, particularly learning by trial and error [2]. This principle originated from Thorndike's "Law of Effect", which states that actions followed by positive outcomes are more likely to be repeated [2]. The second body of work draws from mathematics and engineering, specifically Richard Bellman's dynamic programming (discussed in 2.2) and Markov Decision Processes (discussed in 2.1.1) to solve optimal control problems [2]. Due to its ability to learn without explicit programming and adapt to uncertain environments, RL is a well-suited approach for industrial control, robotics, and autonomous systems [2].

However, the traditional RL was effective for solving problems which are confined to a small, well-defined environment, and the number of possible states is limited [2]. To overcome this limitation, more effective methods are introduced as discussed in the following chapters.

1.2 Deep Reinforcement Learning (DRL)

Traditional RL methods are tabular in nature, they represent the value function as lookup tables that store estimates for each state-action pair [2]. This was useful in small and well-defined environments. Applying this method to real-world problems creates two fundamental limitations.

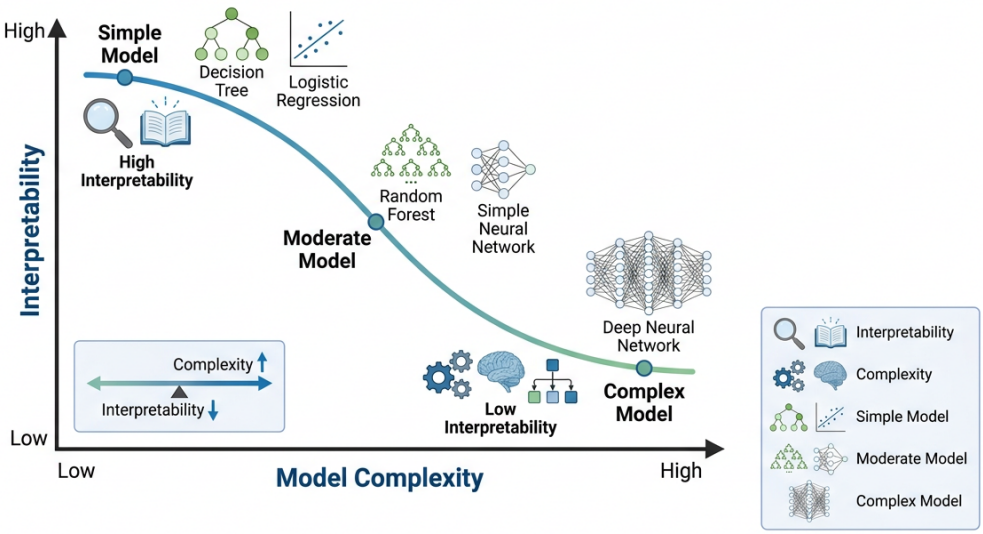


Figure 1.1: Model complexity vs interpretability trade-off

The first limitation is that the continuous state and action spaces must be discretised into bins, which results in discretisation errors, thereby affecting the quality of the solution. The second limitation is that refining the discretisation grid to reduce this error leads to an exponential increase in the number of bins, making the problem completely intractable.

Artificial Neural Networks (ANNs) were introduced as function approximators to solve these limitations, which replaced the lookup table with a parameterised function capable of generalising across the state-action space [3]. This generalisation capability is precisely what makes the "deep" in Deep Reinforcement Learning (DRL) powerful. This approach is significant as most real-world control problems are inherently continuous in nature, both in state space and action space, making the tabular approach ineffective. A landmark demonstration of DRL's capability was the use of a Deep Q-Network (DQN) to play Atari games at an expert level, showing that deep function approximation could scale RL to high-dimensional, complex environments [4].

1.3 Interpretability in DRL

ML models such as linear regression and decision trees are inherently interpretable in their decision logic, making their output easily traceable and understandable to humans [1]. However, as the model complexity increases, the interpretability reduces, since the complexity of the model is inversely proportional to interpretability [5]. Solving complex real-world tasks requires complex models, which in turn sacrifice human interpretability [1], [5] (Figure 1.1).

Consider the difference between a linear regression model and a neural network. A linear regression model, which uses stochastic gradient descent to optimise is easily interpretable; its coefficients directly indicate the contribution of each variable, allowing humans to easily interpret and predict output [1]. However, using a simple, easily interpretable model to solve complex tasks results in poor performance. A classic example is like solving the XOR problem, a single-layer perceptron being interpretable cannot solve XOR because it can only divide the input space with a single linear decision boundary [6]. Another limitation is that linear regression assumes a linear relationship between inputs and outputs, which prevents it from capturing nonlinear patterns.

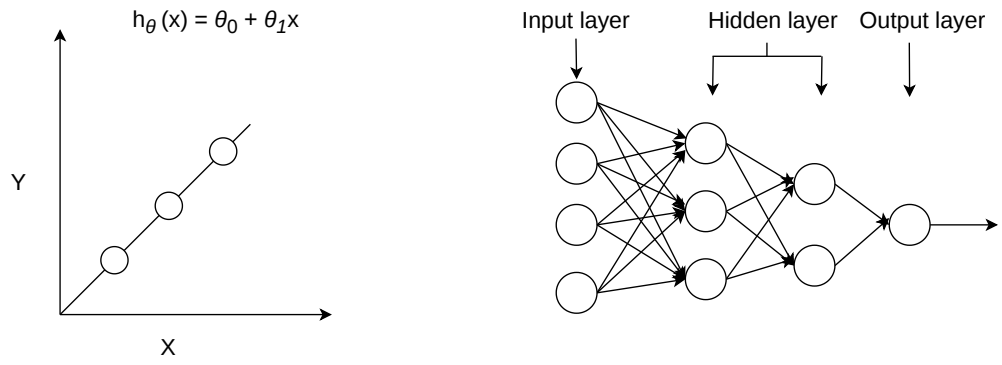


Figure 1.2: Comparison of a Linear Regression model (left) and a Multi-Layer Perceptron (right), illustrating the trade-off between model complexity and interpretability. From linear regression, humans can understand the relationship between the X and Y axis with the help of the straight line, whereas in the multi-layer perceptron, it is difficult to understand the internal working of the model.

A Multi-Layer Perceptron (MLP) works by multiple layers of weights, biases and activation functions. These models are optimised by back propagation (calculating how much the weights contribute to error and then adjusting them to improve future predictions) [1]. These layers create a virtual mapping between input and output, which is harder to interpret since the interaction between these components is non-linear. However, these complex models solve problems which are not easier for human computation using multidimensional matrix multiplication. Since these models are using complex multidimensional matrix multiplication, it creates a black box which is incomprehensible to humans, as shown in the figure 1.2.

Although DRL was an improved version of traditional RL, it showed the black box nature because of the neural networks at its core [7]. This black box nature created the following limitations:

- **Interpretability** - the policies learned by DRL agents are complex and not transparent, making it nearly impossible for humans to understand the internal mechanism of the model [7].
- **Explainability** - the output generated by DRL agents is the result of successive matrix multiplication across multiple layers, which makes it incomprehensible for humans to understand and justify the decision-making process [7], [8].
- **Generalisation** - In terms of generalisation, DRL agents are better than traditional RL agents, but they still underperform and are not reliable in unseen environments or situations [7].

Without proper interpretability, the internal workings and decision-making process of the model cannot be studied or verified, which in turn hinders the optimisation, debugging, and adaptation of the model. This results in the user losing trust in the controller.

1.4 Problem Statement

To address the interpretability and explainability limitations of DRL, several methods have been introduced. A recent systematic literature study by Bekkemoen on Explainable reinforcement learning (XRL) organises these methods into a taxonomy of three main groups [9]

- **Interpretable Agent (IA)** - These methods achieve interpretability by representing the agent in a simple, easily comprehensible way. The representation can be rule-based systems, decision trees, or domain-specific languages [9].
- **Intrinsic Explainability (IE)** - methods involve modifying the model to make it explainable. [9].
- **Post Hoc Explainability (PHE)** - involves methods that provide the model the ability to explain without modifying it[[9].

In the case of system controllers, representing policies in domain-specific languages is effective. These are controller representations expressed as human-readable programs, which significantly improve the interpretability [7], [10], [11]. To aid this approach, a new solution was introduced by Trivedi et al., known as Learning Embeddings for Latent Program Synthesis (LEAPS), which demonstrated promising results in generating interpretable programmatic policies [7]. Although LEAPS addressed the limitations of programmatic policies in complex behaviours, it was confined to discrete environments, limiting its applicability to real-world continuous control tasks.

In this thesis, we present an extended version of LEAPS, capable of operating in a continuous environment. We modified the LEAPS solution to work on a simple continuous space grid world. The main objective is to investigate the generation of programmatic expressions using a Variational Autoencoder (VAE). The first stage consists of training a VAE model on a dataset of valid DSL programs with the goal of deconstructing and reconstructing the original program. In the second phase, an RL loop guides the search for parameters in the latent space that produce the best-performing program to control the given task. An additional objective is the design of a DSL that better balances expressiveness and explainability.

We start by having a brief introduction to Explainable AI, followed by an overview of Explainable Reinforcement Learning (XRL) and Programmatic Reinforcement Learning (PRL). The literature review describes the current state of Programmatic RL, then explains in detail the LEAPS method, followed by the description of the custom environment used in this project. We then present the program generated by the RL agent, along with a post-processed version of it, to improve the readability. A visual representation of the agent navigating the environment by executing the program is also showcased. Finally, a dedicated section that discusses the methods explored and modified throughout the research, as well as the directions for future work.

Chapter 2

Reinforcement Learning

2.1 What is Reinforcement Learning?

Reinforcement learning (RL) is one of the domain in machine learning for solving sequential decision problems using feedback-informed repetition, in which the agent learns what to do, mapping situations to actions, through direct interaction with the environment to maximise the numerical reward [2]. As Sutton and Barto define it *"A computational approach where an agent tries to maximise the total amount of return it receives while interacting with a complex, uncertain environment"* [2].

In contrast with supervised learning, which relies on a training set of labelled examples, and unsupervised learning, which discovers structure in unlabelled data, an RL agent learns through trial and error [1]. Here, the learner is never explicitly instructed on which actions to take. Instead, it must discover which actions yield the maximum reward by trying them, and the reward can be positive or negative [2]. When an agent takes an action which increases its probability of reaching the goal, it receives a positive reward and actions that reduce its probability result in a negative reward. An action can affect the immediate reward as well as the future rewards, so to maximise this cumulative reward, the agent must exploit the actions it has already found [2]. However, to choose the best action, it needs to explore the environment. The balance between exploration and exploitation is one of the core characteristics of an RL system [2]. The problem RL addresses is inherently a closed-loop because the actions taken by the learning system will directly affect the inputs it receives later, as the agent's primary goal is to maximise the cumulative reward rather than simply finding hidden structures in the data.[2]

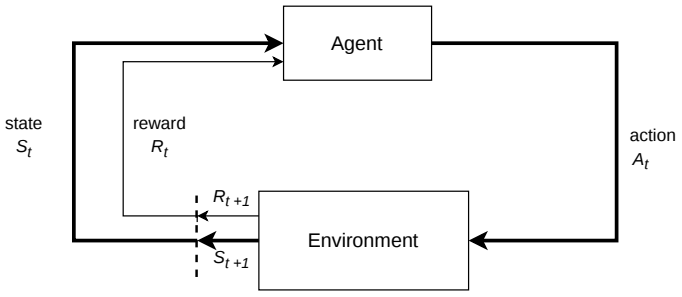


Figure 2.1: An agent-environment interaction in reinforcement learning [2]

The RL system consists of the following elements:

- **Agent** - the learner who interacts with the environment [2].
- **Environment** - a space where the agent interacts and receives feedback from [2].
- **State (s_t)** - a representation of the current situation of the environment observed by the agent at time step t, $s_t \in S$ [2].
- **Action (a_t)** - a decision or move the agent selects from action space A at time step t, $a_t \in A$ [2].
- **Policy (π)** - the behaviour of the agent in a particular state (s_t), or the action (a_t) the agent takes in a state (s_t) [2].
- **Reward Signal ($\mathbf{r}_t$)** - the immediate numerical feedback the agent gets after taking an action, which can be positive or negative [2].
- **Return ($\mathbf{R}_t$)** - it is the cumulative sum of numerical reward signals the agent gets after interacting with the environment [2].
- **Episode (E)** - a finite set of time steps that shows the interaction between the agent and the environment, starting from an initial state and ending at a terminal state. At the end of each episode, the agent is reset back to an initial state [2]. Based on the timescale of the task, RL problems can be classified into two types:
  - **Episodic tasks**: interactions break naturally into episodes [2].

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T \tag{2.1}$$

  - **Continuous tasks**: interaction does not have natural episodes [2].

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

    where γ is the discount rate and $0 \leq \gamma \leq 1$, when $\gamma = 0$ the agent considers only immediate reward. When $\gamma \to 1$, the agent considers future rewards strongly [2].

2.1.1 Markov Decision Process

Before explaining the Markov Decision Process, we should understand what the Markov property is. A system or state signal is considered to have the Markov Property (or Markovian) when the outcome of an action, specifically the transition to the next state and the reward received depend solely on the current state and action and not on prior history of state or action [2]. A reinforcement learning task that satisfies the Markov property is called a *Markov decision process (MDP)* [2]. An MDP is formally defined as a tuple of **(S, A, R, P, γ)**.

- **S** state space
- **A** action space
- **R** reward function
- **P** transition dynamics

- γ - discount factor

When the state and action spaces are finite, it is called a *finite Markov decision process (finite MDP)* [2]. Given any state s and action a, the probability of each possible pair of next state and reward, (s', r) is denoted as [2].

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s,\ A_t = a\} \tag{2.3}$$

and the expected reward for state-action pairs is denoted as.

$$r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \tag{2.4}$$

2.1.2 Value function

A value function estimates how "good" it is for an agent to be in a given state, or perform a specific action in a given state [2]. This goodness is defined as the total amount of reward an agent can expect to accumulate over the future, which depends on the action it chooses [2]. There are two types of value functions.

- **State-value function (v_π)** - It is the expected return starting from the state s and following the policy π [2].

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \tag{2.5}$$

- **Action-value function (q_π)** - It is the expected return starting from the state s, taking action a, and thereafter following the policy π [2].

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s,\ A_t = a\right] \tag{2.6}$$

The *Bellman equation* expresses that the value of a state must equal the sum of expected immediate reward and the discounted value of its successor states [2]. Because the value of a state depends on the value of a state that follows it, which defines a recursive relationship [2]. For any given policy π, the state-value function (v_π) is the unique solution to its Bellman equation [2].

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s\right] \\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right].
\end{aligned} \tag{2.7}$$

A value function defines a partial ordering of the policies, a policy π is considered better than or equal to another policy π', if its expected return is greater than or equal to the other's for all [2]. There exists at least one policy that is better than or equal to others, which is known as the optimal policy [2]. While there may be multiple optimal policies, they all share the same optimal value functions [2]. There are two types of optimal value functions:

- **Optimal state-value function (v^*)** - gives the maximum expected return achievable from a specific state s across all possible policies [2].

$$v^*(s) = \max_\pi v_\pi(s) \tag{2.8}$$

- **Optimal action-value function q^*** - gives the expected return of taking an action a in a specific state s and thereafter following an optimal policy [2].

$$q^*(s,a) = \max_\pi q_\pi(s,a) \tag{2.9}$$

The *Bellman optimality equation* expresses that the value of a state under an optimal policy must equal the expected return for taking the best optimal action from that state [2].

Bellman optimality equation for v^* expressed as

$$\begin{aligned} v^*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s,a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t \mid S_t = s,\ A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s,\ A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s,\ A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s,\ A_t = a\right] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v^*(s')\right]. \end{aligned} \tag{2.10}$$

and Bellman optimality equation for q^* expressed as

$$\begin{aligned} q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') \,\middle|\, S_t = s,\ A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q^*(s', a')\right]. \end{aligned} \tag{2.11}$$

2.2 Temporal Difference

When solving the RL problems, we need a mechanism to calculate the value functions v_π. The two methods for estimating these values are *Monte Carlo Methods (MC)* and *Dynamic Programming (DP)* [2].

Monte Carlo Methods

In MC, the RL problems are solved by averaging the sample returns. These methods estimate the value of a state simply by averaging the complete returns observed after the state is visited [2]. The benefit of this method is that no prior knowledge is necessary.

Dynamic Programming
When a complete and accurate model of the environment as an MDP is provided to the agents, dynamic programming can be used to compute the optimal policies [2]. It is basically a collection of algorithms, it uses value functions to organise and structure the search for good policies [2]. Bellman equations are used to update rules that iteratively improve the approximations of desired value functions. The requirement of a perfect model is one of the key characteristics of DP [2].

TD
The combination of Monte Carlo and dynamic programming (DP) to learn a value function is called *Temporal Difference* (TD). In TD methods similar to MC methods, learning occurs from raw experience without a model of the environment [2]. Similarly to DP, TD methods update estimates based on other learned estimates without waiting for an outcome [2].

The estimation of v_π at a given non-terminal state S_t is based on the cumulative return after visiting that state. In MC, the value is only updated after the complete return G_t is known, which means waiting until the terminal state is reached [2]. This can be formulated as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right] \tag{2.12}$$

with G_t as the actual return at time t, and α is a constant step-size parameter. This method is known as constant-α MC [2]. However, TD methods can calculate the new value as soon as information about the next step is available [2].

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \tag{2.13}$$

Since the TD method bases its update in part on an existing estimate, it's similar to the bootstrapping method, like DP [2].

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.14}$$

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|\, S_t = s \right]. \end{aligned} \tag{2.15}$$

MC methods use (2.14) as the estimate target, whereas Dynamic Programming (DP) methods use (2.15) as the estimate target. The Temporal Difference (TD) method also uses (2.15) as a target, but combines the properties of MC and DP. Similar to MC, TD does not need a model of the environment, it directly samples from the environment. Similar to DP, TD uses bootstrapping, it bootstraps by updating estimates based on the current estimate of $v(s_{t+1})$ [2]. In MC, the target is an estimate because the expected value is unknown. Similarly, in DP, the target is also an estimate, but because the $v_\pi(s_{t+1})$ is not known and the current estimate $v(s_{t+1})$ is used instead [2]. Through this way, TD becomes an intersection between MC and DP.

To compare TD and Sutton and Barto provided a simple analogy [2]. Consider the travel time between work place and home to be 30 minutes. After 10 minutes, a heavy traffic jam is encountered, resulting in an increase in the estimated travel time. In the MC method, after reaching home, the estimated travel time is updated, whereas in the TD method, when a traffic jam is encountered, the initial estimate is updated with a higher estimate. Here, based on the temporal difference between the estimates, learning and updating occur in real-time.

The main advantage of the TD method is that it does not need a model of the environment. Since TD methods do not wait for the episode to end, they can be executed dynamically in real-time. This property makes TD methods highly applicable in continuous tasks, and it requires less memory and computation compared to MC [2].

2.3 RL Methods

At the highest level, RL methods can be broadly classified into model-based and model-free approaches. Within model-free methods, algorithms are further classified as either on-policy or off-policy.[12].

2.3.1 Model-free & Model-based methods

In the model-free method, the agent relies on estimating the value function or policy through a large number of environment interaction samples [12]. The goal is to optimise the action policy to obtain maximum reward for performing a given action in a specific state. Whereas in model-based methods, the agent learns transition dynamics of the environment, known as the "models", and uses this model to plan and choose actions [12]. Compared with model-based methods, model-free methods are generally easier to implement as they rely on experience-based learning.

2.3.2 Off-Policy & On-Policy methods

Off-policy and on-policy learning are methodologies primarily under model-free learning approaches, not depending on environment transition probabilities. In the off-policy method, the algorithm separates the behaviour policy from the target policy [12]. It uses a behaviour policy and interacts with the environment and collects experience, while the target policy is optimised towards the best possible behaviour [12]. The past experience can be reused using a replay buffer, which makes it more sample-efficient.

In the on-policy method, the algorithm evaluates and improves the same policy that it is currently using to interact with the environment. Here, the behaviour policy and target policy are identical, meaning the exploration and learning are tightly coupled [12].

2.4 RL Algorithms

The RL algorithms are mainly classified into three categories based on what they optimise [12]:

- **Value-based methods** - These RL algorithms estimate the expected return of taking an action in a given state by learning an action-value function [12]. Normally, the optimal policy is derived by choosing the action with the highest cumulative reward (figure 2.2).

  e.g. Deep Q-Networks (DQN) explained in section 3.1.1

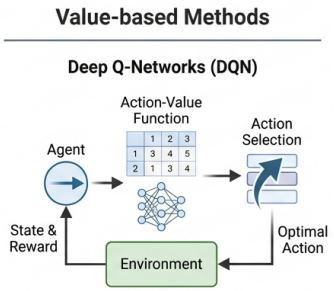


Figure 2.2: DQN algorithm

- **Policy-based methods** - These are RL methods that directly parameterise and optimise the policy and skip the learning of a value function. To improve the agent's performance, the policy parameters are adjusted in the direction of the expected reward (figure 2.3) [12].

  e.g. Proximal Policy Optimisation (PPO) explained in section 3.1.2

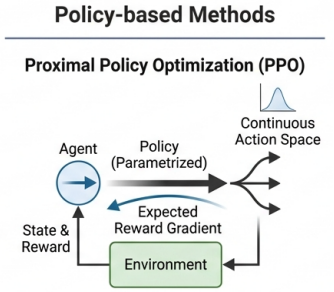


Figure 2.3: PPO algorithm

- **Actor-Critic methods** - It is a hybrid of value-based and policy-based methods [12]. The stability of value-based methods and the efficiency of policy-based methods are combined in this method [12]. Based on a learned policy, the Actor selects an action [12]. The critic then evaluates the action selected by the Actor using a value function [12]. The feedback from the Critic drives the Actor's policy update (figure 2.4).

  e.g. Asynchronous Advantage Actor-Critic (A3C) explained in section 3.1.3

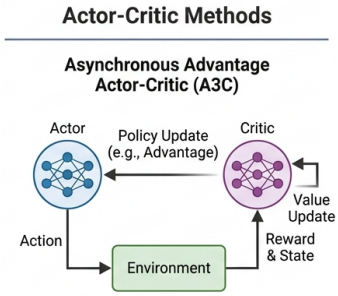


Figure 2.4: A3C algorithm

Chapter 3

Deep Reinforcement Learning

Deep RL is a combination of RL and deep learning [13]. It has achieved massive success in multiple domains such as healthcare, robotics, investment, etc [13].

In traditional RL, the research was mainly based on tabular methods or relatively small-scale function approximators such as linear models with carefully hand-engineered features [13], [14]. These methods excelled in less complex problems. But when complex problems like high-dimensional or continuous state spaces were encountered, these methods were not effective. Deep neural networks, known for their ability to learn complex relationships directly from raw data, offered a way to overcome these limitations [13], [15], [16].

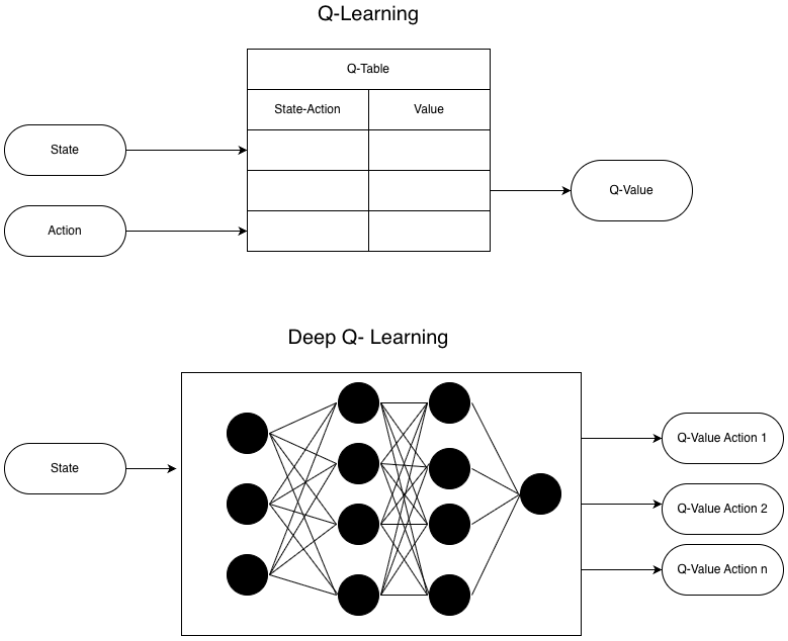


Figure 3.1: A conceptual comparison of tabular Q-Learning (top) vs deep Q-Learning (bottom)[13]

The fig 3.1 shows the comparison between classical Q-learning and deep learning. In traditional Q-learning, each state-action pair has a unique entry in a Q-table, which can quickly

become ineffective for high-dimensional or continuous state spaces. Deep Q-learning uses a neural network to approximate the Q-function, mapping raw state observations directly to estimated Q-values for each action [13].

An important moment in deep reinforcement learning was the introduction of DQN by the researchers at DeepMind [4]. The potential of combining of Q-learning and convolutional neural networks was showcased by defeating human records in dozens of Atari games [4]. However, training a deep neural network to estimate action-value functions *Q(s,a)* introduced two key issues [13]:

- Correlated data: Consecutive samples from the environment are highly correlated; training on them sequentially can cause gradient descent to perform poorly [13].
- Non-stationary targets: Value-based methods like Q-learning update their estimates based on the current estimate of the Q-function itself [13]. Since the network's parameters change continuously during training, it creates a "moving target" problem where the updates are constantly chasing a target that shifts with every single training step [13].

DQN solved these issues with two stabilisation techniques:

- **Experience replay**: Instead of updating the network using only the most recent interaction, the agent stores its past transitions in a replay buffer [13]. While training, the agent randomly samples mini-batches from this buffer, which breaks the correlation found in sequential observations [13].
- **Target networks**: To solve the non-stationary target issue in traditional Q-learning, DQN uses a target network, which is a separate copy of the main Q-network whose weights are completely fixed for a set of iterations [13]. It is only periodically updated to match the latest weights of the main network.

3.1 Algorithms

DRL algorithms are classified into three main categories, based on their learning approach, such as value-based methods, which learn a value function to derive a policy, policy-based methods which directly optimise the policy, and hybrid actor-critic which combines both approaches [13]. In this section, we explore one of the major examples in each approach.

3.1.1 Value-based: Deep Q-Learning

Deep Q-Learning or Deep Q-Network is a value-based RL method. It estimates the long-term return in future when a specific action is taken in a given state [13]. To process high-dimensional inputs, deep Convolutional Neural Networks (CNNs) are used at the core of DQN [13]. Raw state observations are passed through the neural network to estimate the Q-value output for each possible action. To stabilise the training, experience replay and target networks are used in DQN [13]. Experience replay refers to storing the agent's previous experiences and utilising them later, and Target networks are used to prevent the agent from chasing the constantly changing target, thereby stabilising the learning process [17]. DQN potential was showcased when it beat humans in Atari games [17]. To minimise the loss function, DQN uses the gradient descent algorithm, which is formulated as [13]:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot),\, s'\sim\mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \tag{3.1}$$

Algorithm 1 Deep Q-Learning with Experience Replay [18]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
for episode $= 1$ to M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ to T **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \arg\max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in environment and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation (3.1)
  end for
end for

3.1.2 Policy-Based: Proximal Policy Optimization

Policy-based methods involve changing the direction of policy parameters to improve the agent's overall performance. Trust Region Policy Optimization (TRPO) is a policy gradient method which controls the change in policy by using a mathematical constraint called "trust region" [13]. This mechanism prevents the Neural Networks from performing large updates [13]. While TRPO is highly stable, it requires heavy mathematical calculations, making it computationally expensive [13].

To solve the heavy computation of TRPO, Proximal Policy Optimization (PPO) was introduced [13]. It was developed as a light and scalable version of TRPO. PPO achieved the same objective of preventing large changes by using an objective function called **clipped probability ratio** [13]. This ratio compares old and new policies and calculates the chance of selecting an action. This helps in creating stable and controlled updates, deciding first-order objective optimisation without computational overhead [13]. PPO's reliability and scalability make it widely used in complex environments like robotics and multi-agent games [13].

Algorithm 2 PPO, Actor-Critic Style [19]

for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in the environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate objective L with respect to θ, using K epochs and minibatch size $M \leq NT$
  Update old policy parameters: $\theta_{\text{old}} \leftarrow \theta$
end for

3.1.3 Actor-Critic methods: Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor-Critic (A3C) is a DRL algorithm which intends to solve the sample inefficiency and data correlation problems by using a parallel architecture [13]. It combines the power of two approaches. An "actor" network parameterises policy and generates output, and in parallel, the "critic" network estimates the value function and guides the actor updates [13]. Instead of using a single agent, A3C uses multiple agents, and each agent has its own copy of the environment [13]. Each agent explores its own environment and computes gradients for its respective actor and critic locally [13]. These updates are then applied to a global master model at regular intervals. As multiple agents are interacting with their own environment in parallel, the collected experience data becomes decorrelated naturally [13]. Since the data is already decorrelated across multiple agents, the use of a memory-expensive replay buffer is not needed [13].

Algorithm 3 Asynchronous One-Step Q-Learning - pseudocode for each actor-learner thread. [20]

// **Assume global shared parameters** θ**,** θ^-**, and counter** $T = 0$
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  Set $y = \begin{cases} r, & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-), & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients with respect to θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  Update state: $s \leftarrow s'$
  Increment counters: $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \bmod I_{\text{target}} = 0$ **then**
    Update the target network: $\theta^- \leftarrow \theta$
  end if
  if $t \bmod I_{\text{AsyncUpdate}} = 0$ **or** s is terminal **then**
    Perform asynchronous update of θ using $d\theta$
    Clear gradients: $d\theta \leftarrow 0$
  end if
until $T > T_{\max}$

Chapter 4

Explainable Artificial Intelligence (XAI)

To solve real-world problems with high dimensions, the use of complex models like an Artificial Neural Network (ANN) is inevitable. The problem with this approach is the explainability of the models. For instance, when a customer applies for a credit card at a bank and gets the credit card with a certain limit, they have the right to question the decision behind setting the limit to a particular value. If the bank representative replies by stating that their ML model sets the limit based on certain weights and biases, this would likely affect the customer's trust in the bank.

The importance of explainability is further reinforced by the laws as well, notably the launch of the European Union's General Data Protection Regulation, which introduces the right to explanation of all automated decisions for individuals [9], [21].

4.1 What is Explainability

There is a common confusion between interpretability and explainability [8]. Interpretability refers to the internal characteristic of a model. It means the level at which a human can understand just by observing the working of the model [8]. Interpretability is also known as transparency [8]. Whereas explainability means an external characteristic of a model [8]. The model takes any action with the intention of making the target audience understand its internal functions [8]. The following section summarises the most commonly used nomenclature in XAI communities [8]:

- **Understandability**: It denotes the property of a model to make a human understand its function, without explaining its internal structure [8], [22].
- **Comprehensibility**: It refers to the ability of a learning algorithm to represent its knowledge in a form that is understandable to humans [8], [23], [24], [25].
- **Interpretability**: It is defined as the ability to explain or to provide meaning in terms that are understandable to humans [8].
- **Explainability**: It is associated with the concept of explanation as an interface between humans and a decision maker [8], [26].
- **Transparency**: A model is said to be transparent if it is understandable [8], [27].

4.2 XAI

According to Bekkemoen, XAI is defined as an AI that can be understood by a human, including how it works, its strengths and weaknesses, and the behaviour it will exhibit in unseen situations [8]. In contrast, a black box is a system where the internal mechanisms are incomprehensible or inaccessible to humans. The term XAI was popularised by DARPA with their 4-year XAI research program, started in 2017 [28]. The target of this program was an end user who depends on the results, decisions, or recommendations provided by an AI system, and has the right to understand the system's internal steps behind these results [8], [28].

To get more clarity, the definition of XAI given by D.Gunning as: "*XAI will create a suite of machine learning techniques that enable human users to understand, appropriately trust, and effectively manage the emerging generation of artificially intelligent partners*" [28].

4.3 Goals of XAI

The goals of XAI differ with the targeted audience; for example, for some audiences, the goal is trustworthiness, for some, it's fairness [8]. The following taxonomy (4.1) by Arrieta demonstrates the goals of XAI with the respective target audience [8].

XAI Goal	**Main target audience**
Trustworthiness	Domain experts, users of the model affected by decisions
Causality	Domain experts, managers and executive board members, and regulatory entities/agencies
Transferability	Domain experts, data scientists
Informativeness	All users
Confidence	Domain experts, developers, managers, regulatory entities/agencies
Fairness	Users affected by model decisions, regulatory entities/agencies
Accessibility	Product owners, managers, users affected by model decisions
Interactivity	Domain experts, users affected by model decisions
Privacy awareness	Users affected by model decisions, regulatory entities/agencies

Table 4.1: XAI goals and their main target audience. [8]

- **Trustworthiness**: It is the confidence in whether a model will act as expected when facing an unseen problem [8].
- **Causality**: It is the ability of an explainable ML model to provide a relationship between data to validate its results [8].
- **Transferability**: The understanding of the internal mechanism of a model allows the user to reuse this knowledge in another problem statement [8].

- **Informativeness**: Information is crucial for connecting the user's decision to the model's solution [8].
- **Confidence**: An explainable model should contain information about the confidence of its working to provide a generalised, robust and stable solution [8].
- **Fairness**: Since algorithms and models are getting applied in all fields, explainability should be considered as a bridge to avoid unfair or unethical use of an algorithm's outputs [8].
- **Accessibility**: Explainable models will help non-technical and non-expert users to have a better understanding of the model, which helps the users to learn from the machines[8].
- **Interactivity**: This is important where the end users are important and have the ability to modify and interact with the models [8].
- **Privacy awareness**: The ability to evaluate and manage the privacy of data within machine learning systems[8].

All these goals of XAI vary with the target audience. Choosing the correct XAI model which serves the target audience is crucial.

4.4 Methods in XAI

Different types of methods have been proposed for dealing with a variety of data and model types. Among these methods, SHapley Additive exPlanations (SHAP) and Local Interpretable Model Agnostic Explanation (LIME) are the most popular XAI methods [29]. The table (4.2) shows a brief comparison of SHAP and LIME [29].

Metric	**SHAP**	**LIME**
Concept	Applies the model as it is	Uses a local model to explain the complex model
Theory	Game theory based feature attribution	Feature perturbation method
Type	Post-hoc, model-agnostic	
Data type	Images, tabular data, and signals	
Explanation scope	Global and local	Local
Collinearity consideration	Not explicitly handled in the original method	Not explicitly handled
Nonlinear decision handling	Depends on the model and SHAP formulation	Approximates locally, can struggle with strong nonlinearities
Computing time	High	Low
Visualization	Waterfall, beeswarm, summary plots	Single explanation plot

Table 4.2: Comparison between SHAP and LIME explanation methods [29]

4.4.1 SHAP

SHAP is a game theory-based post-hoc, model-agnostic XAI method [29]. It can be applied to any ML models. Each feature is represented as a player, and the model outcome is the payoff [29]. It can explain the contribution of each feature in all instances and in specific instances. The SHAP score shows each feature's contribution to the model output [29]. To calculate this score, all combinations between features are considered to cover all cases, even if all features or a subset of features are used in the model [29]. For local explanation and global explanation, SHAP is widely used. However, SHAP has some limitations. First, SHAP values are dependent on the choice of the ML model, the explanation can vary with the choice of the model [29]. Second, the SHAP scores are misinterpreted [29]. The scores do not highlight the features' actual weight, instead it shows the ranking of the features [29]. Users should give more importance to feature ranking than score value. Third SHAP is vulnerable to biased classifiers, because of it does not show the actual bias, instead it produces superficial explanation which are convincing [29]. Finally, SHAP works by considering that features are independent and variables do not have any correlation, which may not work in actual datasets [29].

4.4.2 LIME

LIME is a post-hoc model-agnostic local explanation method [29], [30]. It explains the influence of each feature in the prediction of a single instance [29]. Each feature's contribution to different classes is visualised using plots [29]. Features are considered independent in LIME and cannot handle the non-linear relationship between features [29]. The complex model behaviour is explained with a local linear model by approximating it and by creating a local dataset around a selected instance, and then reports the feature weights [29]. The feature weights mean that, whenever a feature increases by one unit, provided the other features stay unchanged, how much difference will be there in the model outcome [29]. However, the LIME explanations are dependent on specific instances [29]. Therefore, the LIME explanation cannot explain the entire model behaviour. Similar to SHAP, biased classifiers affect the LIME output. As a result, it generates convincing results which do not project the underlying biases [29].

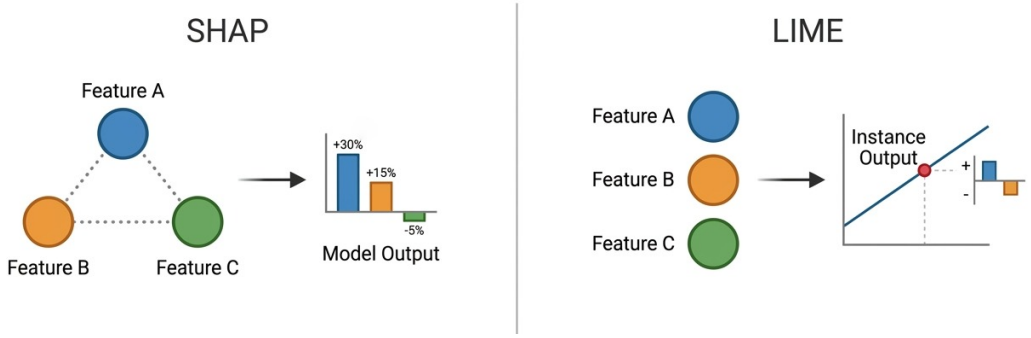


Figure 4.1: On the left, SHAP explains globally and locally based on feature combinations by generating feature contribution rankings. On the right, LIME explains by approximating a local linear model and generating feature influence weights.

Chapter 5

Explainable Reinforcement Learning (XRL)

Explainable Reinforcement Learning (XRL) is a subfield of XAI that focuses on making the decision-making process of RL agents understandable to humans [9]. Since RL agents learn by maximising cumulative returns through interaction, their decisions involve delayed rewards and both short-term and long-term consequences [9]. Therefore, XRL methods must explain immediate, single-step decisions, they must make the targeted audience understand the sequential nature of the agent's behaviour [9]. It should explain how actions influence future outcomes, and how the agent interacts with the environment [9].

According to Bekkemoen's taxonomy, XRL methods can be classified into three categories as shown in Figure 5.1 [9]:

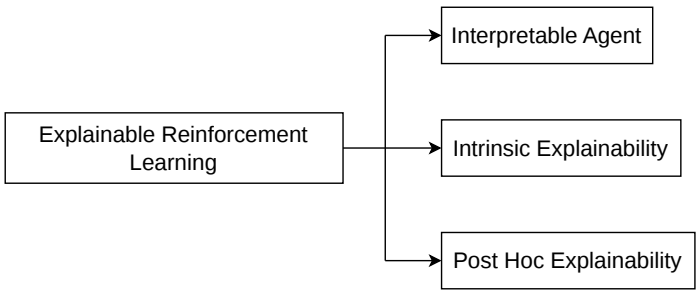


Figure 5.1: Taxonomy of XRL methods [9]

.

- **Interpretable agent (IA)**: It refers to the agent being easily comprehensible and provides an understanding of the underlying relationships [9].
- **Intrinsic explainability (IE)**: It involves methods that modify the RL system before training to make it explainable[9].
- **Post hoc Explainability (PHE)**: It is similar to intrinsic explainability but provides the RL with the ability to explain without modifying it. PHE is applied after training to an unmodified agent [9].

5.1 Interpretable agent

This category consists of agents which are inherently understandable to humans [9]. It does not require any modifications to the agent to be interpretable. This method achieves interpretability by choosing simple function approximators to represent the agent [9]. The explanation in this method is the agent's representation itself (figure 5.2). For instance, when an agent is represented using a decision tree, that decision tree acts as the explanation [9]. Their functional form allows stakeholders to inspect and understand them out of the box [9].

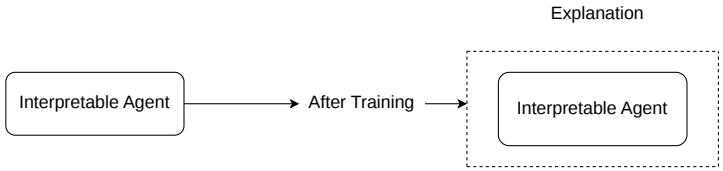


Figure 5.2: The interpretable agent. The explanation is the agent itself [9]

There are five types of interpretable agent methods as shown in Figure 5.3 [9] :

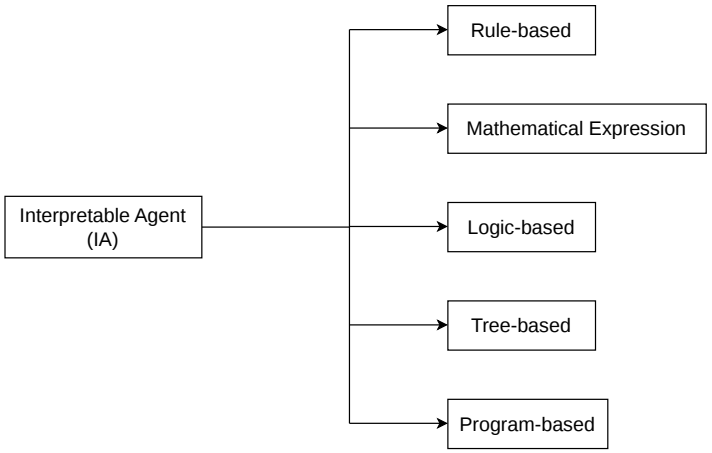


Figure 5.3: Types of Interpretable Agent methods [9]

- **Rule-based**: human-readable rules are used to express the agents.
- **Mathematical expression**: mathematical formulae are used to express the agents.
- **Logic-based**: logical expressions are used to represent the agent's behaviour
- **Tree-based**: tree-based models, such as decision trees, are used to express the agent, where each node corresponds to a decision condition
- **Program-based**: policies are represented as a structured program in domain-specific languages

5.2 Intrinsic Explainability

The intrinsic explainability method involves modifying the agent or model (or both) to make the RL system explainable (figure 5.4 [9]). Model in this context refers to the transition and reward function [9]. This method provides the agent with the ability to generate explanations and is applied before training [9].

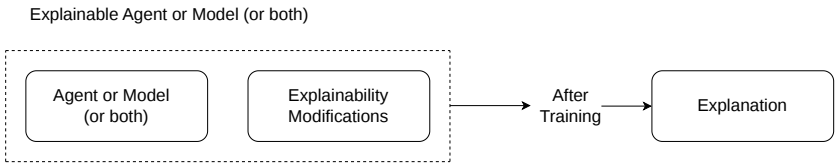


Figure 5.4: The intrinsic explainability approach. Modifies the model to make it explainable[9]

IE methods can be categorised based on how they represent and communicate the explanations, as shown in Figure 5.5 [9].

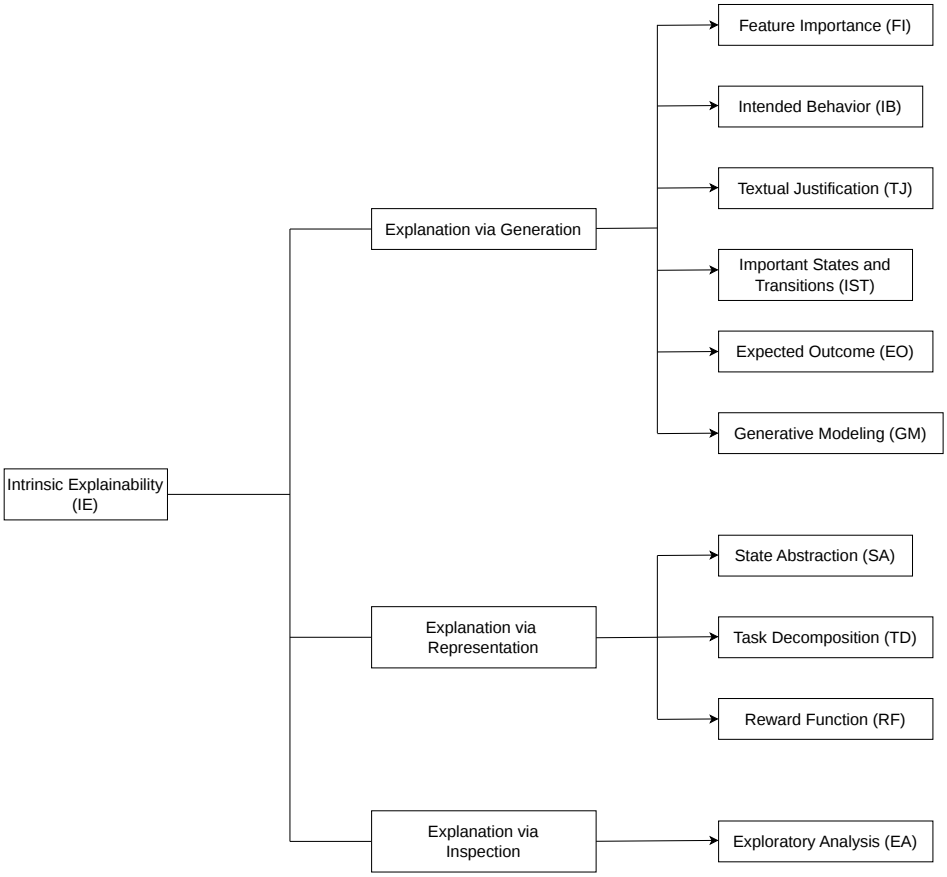


Figure 5.5: Taxonomy of intrinsic explainability [9]

5.2.1 Explanation via generation

In these methods, the agents are modified to generate an explanation through objects [9]. These objects can be a saliency map, a textual response, or any explanatory object [9]. This method can be subdivided into the following [9]:

- **Feature importance**: the agent is modified and uses a saliency map to explain by highlighting features [9].
- **Intended behaviour**: agent's planned actions for the future are informed to the target audience [9].
- **Textual justification**: natural languages in textual form are used by the agent to provide explanation [9].
- **Important states and transitions**: the important states or transition the agent encountered during training is used as explanation [9].
- **Expected outcome**: answers the short-term and long-term consequences of the agent's decision [9].
- **Generative modelling**: generative model approaches are used to understand the agent [9].

The main advantage of these methods are they provide a quicker way to understand an agent's behaviour without heavy computational simulations [9]

5.2.2 Explanation via representation

The agent's representation provides interpretability, such as a decision tree where the decision tree acts as the explanation [9]. Compared to *Explanation via generation*, it does not create any objects to provide an explanation. This method can be subdivided into the following [9]:

- **State abstraction**: state spaces are reduced to make the agent more interpretable [9]. To achieve this, similar states are clustered into abstract states [9]. As the number of states decreases, it results in reducing the situation to be considered for understanding the agent's behaviour [9].
- **Task decomposition**: a task is split into smaller sub-tasks, thereby making the agent interpretable [9]. The divide and conquer method is used to achieve explainability [9].
- **Reward function**: to understand the agent, the reward function is used [9]. The reward function is expressed in an interpretable format through which the target audience can understand the agent's behaviour [9].

5.2.3 Explanation via inspection: exploratory analysis

In this method, the stakeholder needs to inspect, analyse, and assess the explanations manually to extract insights from the agent [9]. Compared to other methods mentioned above, this category is more open-ended, resulting in greater demands on human labour and time [9].

5.3 Post hoc explainability

PHE consists of methods applied to black-box agents or models (or both) [9]. This method aims to extract insight and produce explanations without changing the agent [9]. The main difference between PHE and IE is that PHE is applied after training, whereas IE is applied before training (Figure 5.6) [9].

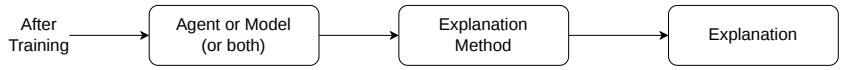


Figure 5.6: Post hoc explainability approach. Extracts information and produce explanation, without modifying the agent[9]

PHE can be categorised into the following, as shown in Figure 5.7 [9]:

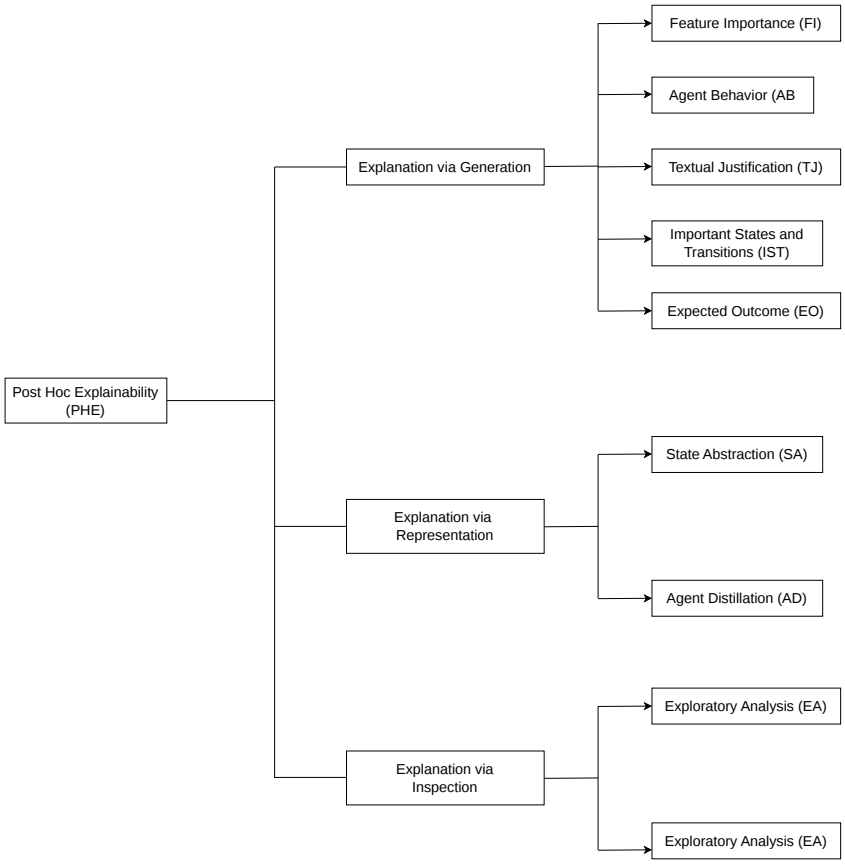


Figure 5.7: Taxonomy of Post hoc explainability [9]

5.3.1 Explanation via generation

In this category, for the tool of explanation, the methods generate objects [9]. It can be visual, textual or some other format [9]. This category can be subdivided into the following [9]:

- **Feature importance**: provides the same explanation as defined in feature importance in IE, explained in section 5.2.1 [9].
- **Agent behaviour**: comprehends the agent by characterising its behaviour [9].
- **Textual justification**: aims to extract explanations in textual form represented in natural language, it is similar to Textual justification in IE, explained in section 5.2.1 [9].
- **Important states and transitions**: aims for the stakeholders to develop a mental model of the agent's behaviour by seeing examples of it in various situations, so that the stakeholders can imagine how the agent will behave in seen and unseen situations, which creates a global understanding [9].
- **Expected outcome**: This method uses transition data to explain what an agent does [9]. This requires the agent to be simulated in the environment, or there should be enough data which can be used to analyse the agent [9].

5.3.2 Explanation via representation

This category uses the internal representation of the agent to explain [9]. These representations are extracted during training. This category can be subdivided into the following [9]:

- **State abstraction**: states are grouped by various similarity measures to reduce the state space complexity, resulting in trading off explanation against complexity [9].
- **Agent distillation**: the agents decision making logic is simplified to do the explanation [9]. To achieve this, the agent is considered an expert and uses imitation learning to learn a distilled agent, which is easier to interpret [9].

5.3.3 Explanation via inspection

These methods are applied to extracting understanding after agent training [9]. This category can be subdivided into the following [9]:

- **Exploratory analysis**: dimension reduction techniques or XAI methods are used to extract the agent behaviour [9].
- **Visual analytics**: to understand, RL agent interactive visualisation and analysis tools are provided [9].

Among all these methods, this thesis work focuses on a program-based interpretable agent approach, specifically programmatic policy synthesis, which forms the foundation of this work, as discussed in the following chapters.

Chapter 6

Related Works

In this section, we will discuss the two main related works for this thesis. The Programmatically Interpretable Reinforcement Learning (PIRL) by Verma et al [31] and LEAPS by Trivedi et al [7], which is the foundation for this thesis work.

6.1 Programmatically Interpretable Reinforcement Learning (PIRL)

Verma et al. developed an RL framework that is designed to generate RL policies using a high-level domain-specific programming language that is human-readable, interpretable and formally verifiable [31]. This approach addresses the black box nature of DRL which makes them difficult to trust or verify in safety-critical applications [31].

6.1.1 Framework

PIRL represents the agent's policy using a high-level domain-specific programming language. It models a reinforcement learning setting as a Partially Observable Markov Decision Process (POMDP) [31].

$$M = (S, A, O, T(\cdot \mid s, a), Z(\cdot \mid s), r, \text{Init}, \gamma) \tag{6.1}$$

Here S indicates the set of states, A represents the set of actions and O is the set of observations [31]. When an agent takes an action a at state s changes the state, and the resulting state follows the distribution $T(\cdot \mid s, a)$ [31]. The probability of creating an observation o by the agent at state s is $Z(\cdot \mid s)$ [31]. In state s, the reward the agent receives on choosing an action a is given by *r(s, a)* [31]. Init is the initial state distribution, and $0 < \gamma < 1$ is the real constant that is used to define the agent's cumulative reward over time [31]. For the model M, the history is represented as a sequence $h = (o_0, a_0, \ldots, a_{k-1}, o_k)$, where o_i and a_i are the agent's observation and action at the i-th time step [31]. H_M represents the set of histories in M [31]. The function to map each history to an action a_k is defined by policy π as a function $\pi : H_M \to A$ [31]. The distribution over rewards R_i that the agent receives at the i-th time step is created by the policy [31]. The agent's return under π is given by [31]

$$R(\pi) = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R_i\right] \tag{6.2}$$

A distinctive feature of PIRL is that the policies are expressed in a high-level domain-specific language [31]. The syntax of this language is shown in 6.3. E represents atoms and x represents histories. The semantics of the language are shown below [31].

$$E ::= c \mid x \mid \oplus(E_1, \ldots, E_k) \mid \texttt{peek}(x, i) \mid \texttt{fold}((\lambda x_1, x_2.\ E_1), \alpha) \tag{6.3}$$

- c: It spans over a collection of numerical constants, and $\oplus$ is a basic operator [31].
- **peek(x,i)**: It returns observation from a history x at i-th time step [31].
- **fold**: It is a standard higher-order combinators [31].
- x, x_1, x_2: these are variables, which are assumed to be inputs [31].

Discovering an optimal policy among a large space of legitimate programs is impossible without some prior on the shape of the target policies [31]. PIRL allows the specification of such priors through instance-specific syntactic models called *sketches* [31]. A sketch is obtained by restricting the grammar in 6.3. The syntax mathematically mirrors the behaviour of discrete PID controllers [31]. This is achieved by restricting the syntax to a sketch so that developers can encode their domain knowledge in the system [31]. This results in the final policy changing the shape of a PID controller [31].

6.1.2 Neurally Directed Program Search (NDPS) Algorithm

Finding optimal policies is a standard procedure in RL [2]. The non-smoothness in the search space of programmatic policies is a fundamental challenge in PIRL [31]. NDSP solves this by first creating a neural policy oracle e_N by using a DRL [31]. This high-performing neural policy is uninterpretable. This policy acts as an approximation of the policy we seek [31]. It then performs a search over the space of programmatic policies to find a program whose optimal reward minimises the distance to the neural policy oracle's output for the given environment [31]. During this process, the programmatic policy may enter states that the original neural network never encountered by mistake [31]. To solve this issue, NDPS simulate the current program to get new states and checks what action the oracle would have taken in those states and adds these new data to the training loop [31].

NDPS Algorithm Details

- `create_histories` - sample set of histories of e_N [31].
- `initialize` - generate starting point program for the policy search [31].
- `collect_reward` - expected aggregate reward calculation [31].
- `update_histories` - actions from the oracle for those interesting inputs in the trajectory of the learned program are obtained [31].
- `neighborhood_pool` - generates a space of programs that are similar to e [31].

Algorithm 4 Neurally Directed Program Search [31]

Input: POMDP M, neural policy $e_{\mathcal{N}}$, sketch $\mathcal{S}$
$\mathcal{H} \leftarrow$ `create_histories`$(e_{\mathcal{N}}, M)$
$e \leftarrow$ `initialize`$(e_{\mathcal{N}}, \mathcal{H}, M, \mathcal{S})$
$R \leftarrow$ `collect_reward`(e, M)
repeat
    $(e', R') \leftarrow (e, R)$
    $\mathcal{H} \leftarrow$ `update_histories`$(e, e_{\mathcal{N}}, M, \mathcal{H})$
    $\mathcal{E} \leftarrow$ `neighborhood_pool`(e)
    $e \leftarrow \arg\min_{e' \in \mathcal{E}} \sum_{h \in \mathcal{H}} \|e'(h) - e_{\mathcal{N}}(h)\|$
    $R \leftarrow$ `collect_reward`(e, M)
until $R' \geq R$
Output: e'

6.1.3 Results

Verman. et al tested the PIRL framework and the NDPS algorithm in the *The Open Racing Car Simulator* (TORCS), a continuous car-racing simulator and classic control games like Cart-Pole [31]. While the results didn't show much improvement over the raw DRL models, they showed the following results [31] :

- **Performance**: Although the programmatic policies did not perform as the DRL, it consistently passes significant performance bars, such as completing race laps that non-oracle-guided agents consistently crashed [31].
- **Generalisation (Transfer Learning)**: PIRL policies performed significantly in unseen environments [31].
- **Robustness to noise**: The policies were robust to defective sensor data used in the observation space [31].

6.1.4 Advantages

The PIRL framework and NDPS algorithm offered several advantages, such as:

- **Interpretability**: PIRL policies are represented in a human-readable, domain-specific programming language, which is easier to interpret compared to the black box nature of DRL [31].
- **Formal Verifiability**: Since PIRL policies are structured programs, they are easier for symbolic verification [31].
- **Stability**: The programmatic sketches act as a regulator, resulting in smoother trajectories [31].

6.1.5 Limitations

The PIRL and NDPS had several advantages from the DRL. Some drawbacks also accompanied it. The programmatic policies were interpretable and robust, but not as performant as those generated by DRL. Another limitation is that the framework only considers deterministic policies, and the dependency on *sketch* to find an optimal program creates human dependency [31].

6.2 LEAPS

The programmatic RL framework introduced by Verma et al [31]. had several advantages, but the approach was dependent on a set of predefined program templates, which needed further modifications for each task. The approach proposed by Trivedi et al, the **L**earning **E**mbeddings for l**A**tent **P**rogram **S**ynthesis (LEAPS), uses a two-stage mechanism, learning program embedding stage and latent program search as shown in Figure 6.1 [7]. The first stage involves using a Variational Auto Encoder (VAE) for learning the program embedding space [7]. The second stage uses the Cross Entropy Method (CEM) (a gradient-free continuous search algorithm) to find the best candidate program that maximises return for a specific task [7]. This way of program generation removes the dependency on a set of predefined program templates.

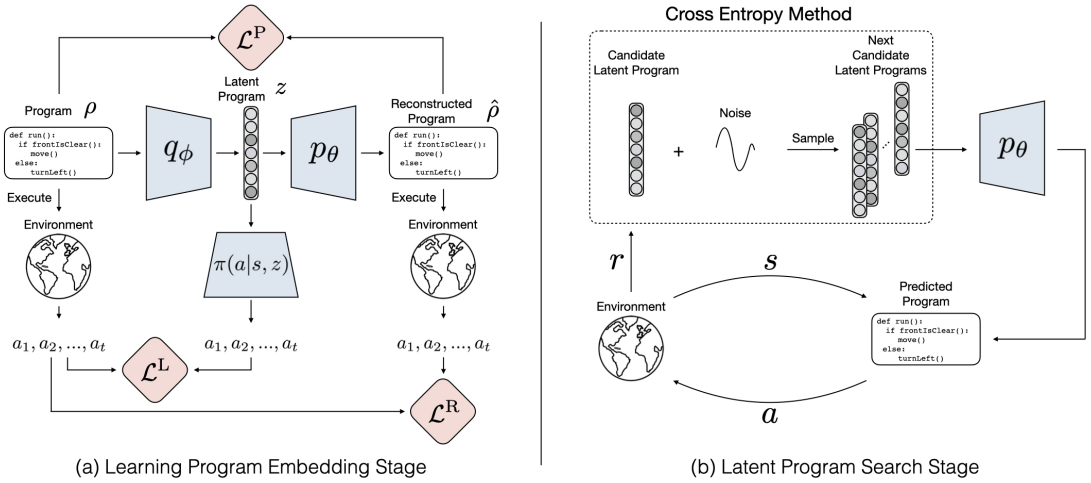


Figure 6.1: Two stages of the LEAPS model, the first stage is learning a program embedding space and the second stage is finding a program with maximum return [7].

6.2.1 Loss functions

A Variational Autoencoder (VAE) is a neural network [32]. It transforms discrete programming code into continuous latent vectors by training on a large dataset of randomly generated programs [7]. It consists of an encoder ($q\phi$) and a decoder ($p\phi$). The encoder takes a program ρ as input and convert into latent program embeddings z. The decoder takes the latent embeddings and outputs the program tokens one by one to reconstruct the original program [7]. In LEAPS, the VAE is trained on on 50,000 unique programs specific to the Karel tasks [7].

Program reconstruction

When a program ρ consisting of a sequence of program tokens is given as input to VAE, the encoder processes the tokens sequentially and outputs the latent program embedding z. The decoder then processes the latent program embedding z and outputs the program tokens to generate a reconstructed program $\hat{\rho}$ [7]. The encoder and decoder are trained to optimise the β-VAE loss (balancing reconstruction accuracy with latent-space regularization) [7], [32]:

$$\mathcal{L}^{P}_{\theta,\phi}(\rho) = -\mathbb{E}_{z\sim q_\phi(z|\rho)}\big[\log p_\theta(\rho \mid z)\big] + \beta\, D_{\mathrm{KL}}\big(q_\phi(z|\rho) \,\|\, p_\theta(z)\big) \tag{6.4}$$

The Kullback-Leibler (KL) divergence, denoted by D_{KL} divergence, which is a mathematical penalty term, is used to calculate the β-VAE loss [7], [32]. In the LEAPS framework, the KL-divergence penalty forces the latent program distribution to shape like a standard normal distribution $\mathcal{N}(0,1)$, resulting in a smoother space rather than isolated points [7], [32].

Program Behaviour Reconstruction

To enforce that programs with the same semantics have similar program embeddings, Trivedi et al. introduced an objective that compares the execution traces of the input and reconstructed programs [7]. This encourages the model to align behaviourally similar and syntactically similar programs to similar latent programs , and it is defined as [7]:

$$\mathcal{L}^{R}_{\theta,\phi}(\rho) = -\mathbb{E}_{z\sim q_\phi(z|\rho)}\left[R_{\text{mat}}(p_\theta(\rho \mid z), \rho)\right] \tag{6.5}$$

Here $R_{\text{mat}}(\hat{\rho}, \rho)$ indicates the reward for matching the input program, it is defined as [7]:

$$R_{\text{mat}}(\hat{\rho}, \rho) = \mathbb{E}_\mu\left[\frac{1}{N}\sum_{t=1}^{N}\mathbf{1}\{\text{EXEC}_i(\hat{\rho}) = \text{EXEC}_i(\rho)\ \forall i = 1, 2, \ldots, t\}\right] \tag{6.6}$$

where N is the maximum length of the programs execution trace and $\text{EXEC}_i(\rho)$ is the action taken by program ρ at time i [7].

Latent Behaviour Reconstruction

To create a smoother behaviour interpolation, another supervision method is devised by learning a program embedding-conditioned policy [7]. It is denoted as $\pi(a \mid z, s_t)$. This recurrent policy takes the program embedding z as input and learns to predict the agent's actions. For an input program ρ, the actions predicted by the policy are defined as [7]:

$$\mathcal{L}\pi^{\text{L}}(\rho, \pi) = -\mathbb{E}\mu\left[\sum_{t=1}^{M}\sum_{i=1}^{|\mathcal{A}|}\mathbb{1}\{\text{EXEC}_i(\hat{\rho}) == \text{EXEC}_i(\rho)\}\log\pi(a_i|z, s_t)\right], \tag{6.7}$$

where M denotes execution length of ρ. Optimising this ensures that similar behaviours are encoded together and allows smooth interpolation [7].

Through these three supervision methods, they optimise to learn a program embedding space which allows for smooth interpolation and can be used for searching for desired agent behaviour [7]. The combined objective is:

$$\min_{\theta,\phi,\pi}\lambda_1\mathcal{L}\theta, \phi^{\text{P}}(\rho) + \lambda_2\mathcal{L}\theta, \phi^{\text{R}}(\rho) + \lambda_3\mathcal{L}^{\text{L}}_{\pi}(\rho, \pi), \tag{6.8}$$

where $\lambda1$, $\lambda2$ and $\lambda3$ controls the importance of each loss. Optimising these parameters makes the program embedding both semantically and syntactically informative [7].

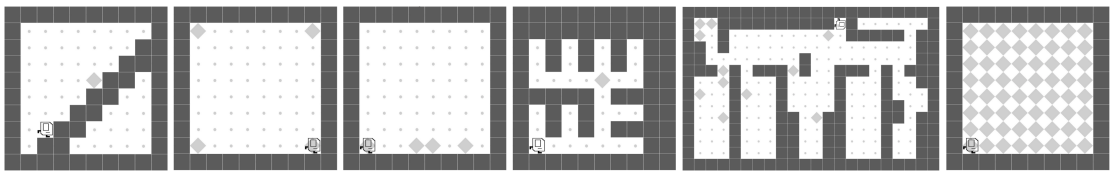
6.2.2 Latent Program Search

CEM search is a gradient-free continuous search algorithm [7]. In LEAPS, instead of generating the code directly it searches for best programs in the learned continuous programming embedding space [7]. It starts by sampling a population of candidate latent programs around a central vector [7]. These candidate programs are then decoded and executed in the environment and gets the rewards [7]. The candidate program with the highest rewards are selected and their weighted average is used to update the center of the sampling distribution [7].

After learning the program embedding space, the next goal is to find a latent program that maximises the reward. To search over the program embedding space, CEM search is [7]. It works by sampling a distribution of latent programs, then decoding sampled latent programs into programs using the learned decoder p_θ [7]. Finally, the programs are executed in the task environment, and the corresponding rewards are obtained [7]. This process is repeated until convergence or the maximum number of steps is reached [7].

6.2.3 Environment

For evaluating the framework, the Karel domain is used. It features an agent navigating through a grid-world consisting of walls and interacting with markers as shown in the Figure 6.2. It consists of 5 actions for moving (`move()`,`turnLeft()`,`turnRight()`,`putMarker()`,`pickMarker()`) and 5 perceptions (e.g `frontIsClear() or markersPresent()`) for detecting obstacles and markers [7]. The walls act as obstacles, and the markers are the objects the agent interacts with, like picking or placing [7]. The programs may contain infinite loops, and the framework handles this scenario by restricting the maximum program execution length to exactly 100 timesteps [7]. For failure scenarios like placing the marker in an incorrect location, the final reward is immediately set to 0 [7]. This prevents the agent from exploiting the reward function by repeatedly placing the markers on every square grid [7].



(a) StairClimber (b) FourCorner (c) TopOff (d) Maze (e) CleanHouse (f) Harvester

Figure 6.2: Karel environments [7]

6.2.4 Results

The LEAPS framework, tested on the Karel environment, demonstrated the best performance compared to traditional DRL. It achieved the following results [7]:

Table 6.1: Rewards on 100×100 grids [7].

Method	STAIRCLIMBER	MAZE
DRL	0.00 (0.00)	0.00 (0.00)
VIPER	0.00 (0.00)	0.10 (0.12)
LEAPS	1.00 (0.00)	1.00 (0.00)

- **Task performance**: LEAPS achieved high performance in all tasks except one. It outperformed in tasks which required repetitive actions, such as climbing stairs or visiting the four corners of a grid, with a perfect mean return of 1.00 [7]. Since it can generate expressive code, it utilises the "while" loops, whereas traditional programming baselines like decision tree (VIPER- a decision tree programmatic policy which imitates the behaviour of a DRL) are incapable of representing the loop behaviour [7], [10]. LEAPS underperformed only in the "Harvester" task, because the optimal program to solve it was longer and syntactically

more complex than the random programs LEAPS trained on and achieved only a mean return of 0.45, whereas the DRL received 0.90 [7].

- **Generalisation**: The LEAPS was trained on small environments, such as 12 x 12 or an 8 x 8 grid [7]. When the agents were tested on massive environments like a 100 x 100 grid, they showed impressive results, outperforming all the DRL and programmatic baselines as shown in table 6.1 [7]. In other words, LEAPS achieved **zero-shot generalisation** (ability of a model or policy to successfully operate in unseen scenarios without any additional training or fine-tuning). This demonstrates the application in multiple scenarios without retraining the agent.

- **Analysis and Debugging**: LEAPS produced explicit human readable code. This resulted in policies which are highly interpretable and interactive. Trivedi et al. experimented with human evaluators having basic programming knowledge to evaluate the programs generated by LEAPS and optimise them with 3 and 5 edits to improve the performance [7]. The program's average reward jumped to 97% and 125%, respectively. This experiment proved that the LEAPS policies can be diagnosed and manually tweaked by humans [7].

6.2.5 Limitations

The LEAPS framework achieved high performance in a variety of areas by outperforming DRL and programmatic baselines [7]. It had some limitations. The complexity of the programs is coupled with the chosen DSL [7]. Another major limitation was that it was designed for a discrete environment, which makes it less applicable for real-world applications since real-world applications are mainly continuous environments and continuous action spaces.

The application of PRL mainly depends on the application. There are scenarios where DRL performed better than LEAPS (Harvester task in the Karel environment). If the performance is weighted over the explainability, then PRL can be considered as a second choice.

Chapter 7

Methodology

Our main contribution to this thesis is the extension of the LEAPS framework to control problems with both continuous state and action spaces. In this thesis, we start from the major limitation of the LEAPS framework, i.e., modifying the framework to adapt it to a continuous environment. The modification includes extending the framework's core architecture, starting with the environment, DSL, VAE, and CEM search as shown in 7.1.

Component	**Traditional LEAPS**	**Modified LEAPS**
DSL	Discrete tokens only like `MOVE, TURN`	Parameterised tokens like `MOVE 0.012, TURN 0.29`.
VAE	Encodes and decodes discrete token-only programs.	Encodes and decodes token + parameter programs.
CEM Search	Searches the latent space for high-return discrete token programs.	PPO-assisted search in latent space for high-return programs.

Table 7.1: Comparison of Traditional LEAPS and Modified LEAPS

7.1 Environment

For this thesis, we developed a custom Gymnasium continuous grid-world environment as shown in Figure 7.2. The environment is designed as a 2D continuous grid with x and y ranges from (0,1).

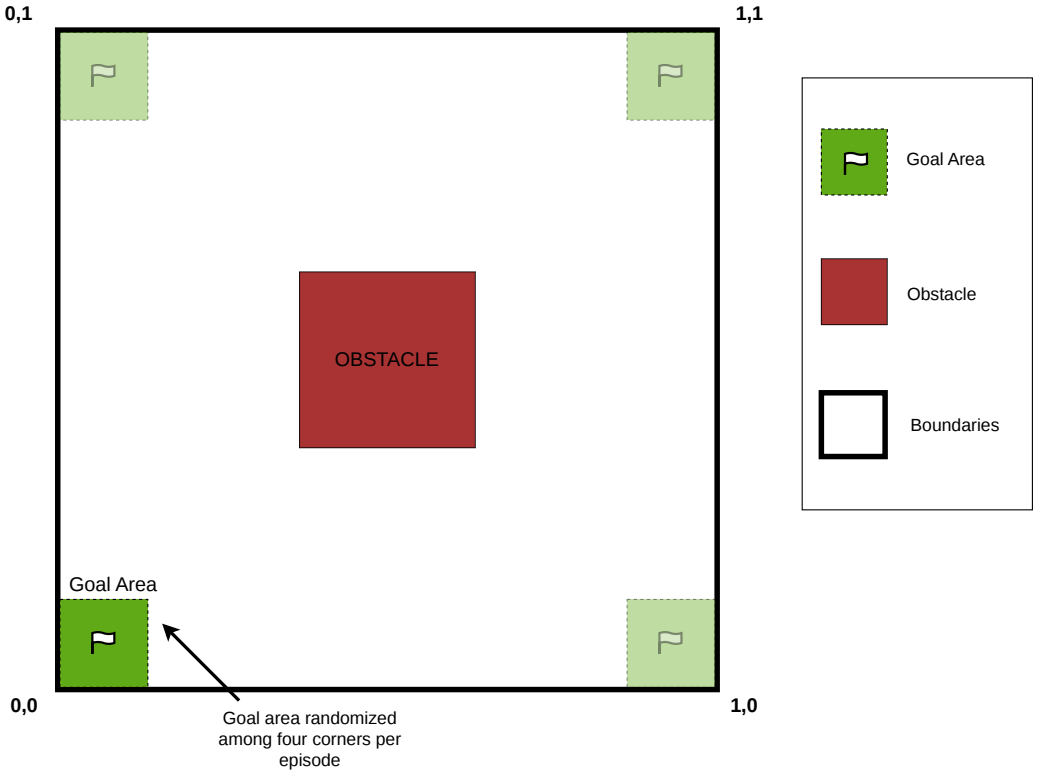


Figure 7.1: Custom continuous environment consisting of a random goal position at four corners and an obstacle at the centre.

The environment was initially designed with a single fixed goal position environment; later extended to a random placement of goal positions at the four corners to mimic the *Fourcorner* environment in Karel tasks. This was made to evaluate whether the agent can generalise multiple goal positions rather than memorising a single trajectory. The environment's properties are defined as follows:

7.1.1 State Representation

The state is represented by three parameters x, y, θ, these are the coordinates and the orientation, respectively. Both the x and y coordinates have continuous values within the range $[0.0, 1.0]$ and the orientation angle θ is bounded between $[-\pi, \pi]$ (the agent can move forward only and can turn -π to π).

7.1.2 Observation Representation

Since this work is an extended version of the traditional LEAPS framework, in which the *Karel* environments are represented as 3D visual tensors (Channels, Height, Width). The environment maps the raw numerical coordinates into a 16-channel pixel-like grid (C=16, H=32, W=32) as shown in the Table 7.2, which makes it compatible for Convolutional Neural Networks (CNN)

processing and matches with the traditional LEAPS framework. The channels encode positions, orientation, the goal's relative offset/angle, distance to obstacle, and boundaries.

Channel	Description
0	Agent x-coordinate
1	Agent y-coordinate
2	$\cos(\theta)$ — orientation encoding
3	$\sin(\theta)$ — orientation encoding
4	Goal region spatial marker
5	Obstacle region spatial marker
6	Normalised distance to goal
7	Goal alignment score (1 = facing goal)
8	Normalised distance to obstacle
9	Normalised distance to nearest border
10–15	Unused

Table 7.2: Overview and description of all channels in the VAE model.

7.1.3 Action Space

The action space consists of two main actions [*move_dist, turn_angle*]. This moves the agent forward by *move_dist* distance and rotates the agent by *turn_angle* radians. These raw values of the action space are converted before execution. The `MOVE` parameter is mapped to (0.05,0.35) by a *sigmoid* function, and the `TURN` parameter is mapped to (-1.0,1.0) by *tanh(x)* function.

Goals & Obstacles

At each episode, the goal is randomly placed at each corner of the square grid, and the obstacle is fixed at the centre of the grid as shown in the Figure 7.2. On entering the obstacle, the episode is terminated, and the agent is awarded a negative reward.

Reward Dynamics

The agent gets the following rewards:

- The agent gets a potential-based reward by calculating the previous distance to the goal and the current distance to the goal, and then scaled by a factor of 10. As the current distance to the goal decreases, the reward increases.
- The agent is rewarded with +50 points when within 0.1 units of the goal, both in x, y and -20 points, and episode termination for crashing into the centre obstacle.
- -2 points is awarded to the agent for bouncing against the map boundary. Initially, termination of the episode was added when hitting the boundary, and later changed to non-termination, which helped the agent recover and increased the overall return.
- Agent gets -0.1 points if it stays idle.

The maximum episode steps, is set to 50; this is designed to bound the maximum distance the agent can travel in an episode. To encourage the agent from avoiding the obstacle, in 30% of the

episodes, the agent is spawned near the obstacle, encouraging the agent to effectively navigate away from the obstacle. The orientation is encoded as *cos(θ) and sin(θ)* for cyclical encoding that avoids the discontinuity at $\pm\pi$ (to make -π and π closer, instead of making it as far away units).

7.1.4 Domain Specific Language (DSL)

A custom DSL was designed for the continuous environment with the following properties:

Category	**Tokens**
Actions	`MOVE, TURN`
Conditions	`NEAR_GOAL, FACING_GOAL, NEAR_OBSTACLE, NEAR_BORDER`
Keywords	`WHILE, IF, ELSE, DO, NOT, END`

Table 7.3: Categories and Corresponding Tokens

- **Actions**: The `MOVE` and `TURN` tokens are accompanied by float parameters to indicate the magnitude of the actions. The parameters for these tokens are bounded. The `MOVE` token parameters are bounded to `[0.05,0.35]`; this prevents the agent from making larger moves or taking big steps and making zero-displacement steps. For the `TURN` token, `tanh` is used to bound it between `[-1,1]` radians, which creates a bounded movement for the agent.

  e.g `MOVE 0.132, TURN -0.172`

- **Conditions**: The boolean logic functions are evaluated using threshold values within predefined ranges, as shown in table 7.4. Each of these tokens has an associated parameter generated by the decoder. These parameters are passed through a bounded sigmoid function to map them into the specified threshold range shown in the table 7.4. This procedure ensures that the decoded values are always meaningful. The threshold values were calculated during the development process.

Evaluators	**Threshold range**	**Description**
`NEAR_GOAL`	[0.10, 0.45]	A conditional token that checks whether the Euclidean distance between the agent's current position and the goal position is less than the threshold.
`FACING_GOAL`	[0.30, 1.20]	Checks whether the agent's current orientation is facing the goal, i.e. the angle difference is less than the threshold.
`NEAR_OBSTACLE`	[0.08, 0.30]	A conditional token that checks whether the Euclidean distance between the agent's current position and the obstacle position is less than the threshold.
`NEAR_BORDER`	[0.08, 0.25]	A conditional token that checks whether the perpendicular distance between the agent's current position and the border is less than the threshold.

Table 7.4: Evaluator predicates and threshold ranges

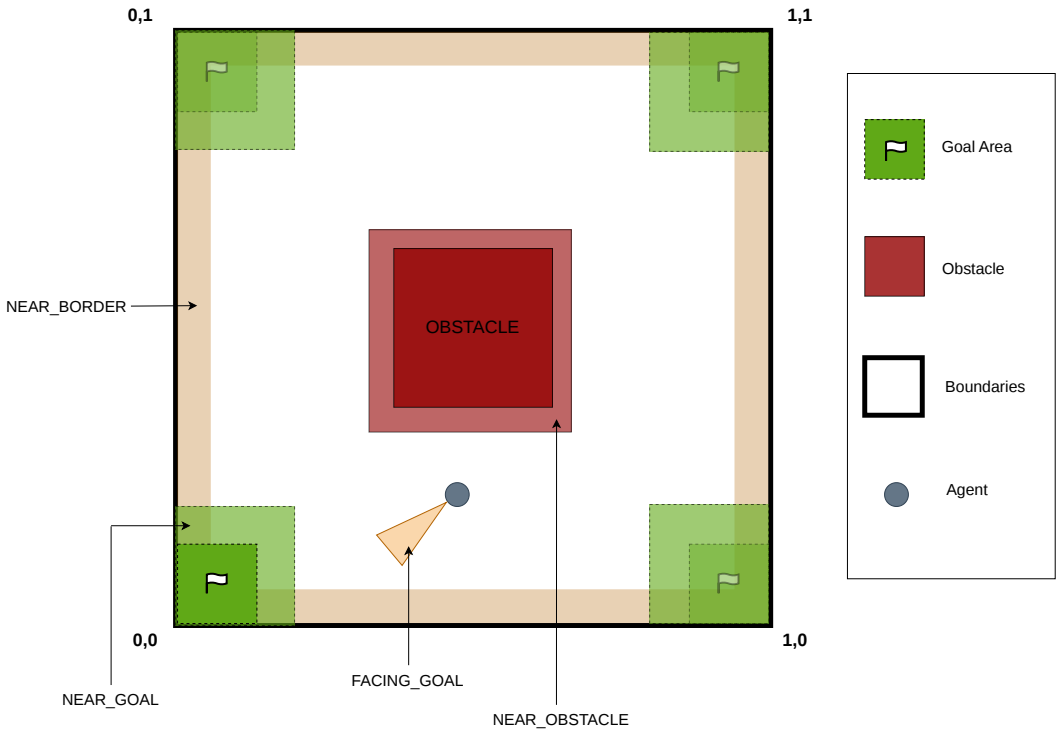


Figure 7.2: Custom environment evaluators depicting `NEAR_GOAL`, `FACING_GOAL`, `NEAR_OBSTACLE`, `NEAR_BORDER`

- **Control Flow**: The control flow enables the generation of syntax for nesting loops and logic. The keywords consist of `WHILE, DO, IF, ELSE, NOT and END`. Some fail-safe mechanisms are added for these tokens. During initial development, the agent was generating chained `NOT` tokens like `NOT NOT NOT NEAR_GOAL`; this was corrected by collapsing chained tokens into one token like `NOT NEAR_GOAL`. For the `WHILE` loop, it is limited to 10 iterations to prevent infinite loops in decoded programs. This 10-iteration limiting was designed after a trial-and-error method. If the value is increased, the agent generates a chain of `MOVE` or `TURN` tokens, resulting in a less optimised program. If the value is decreased, the agent was unable to reach the goal since we capped the `MOVE` and `TURN` to create a safe movement for the agent.

Another important design decision was to track the number of steps inside and outside the loop. This is designed to prevent the agent from generating a sequence of tokens and, at the end, add a control flow to prevent negative rewards, encouraging the agent to use loops and conditions meaningfully. To prevent valid tokens in an invalid position, for example, `NOT` without a condition, the parser is designed to skip these invalid token arrangements and prevent interruption errors during parsing.

7.1.5 Variational Auto Encoder (VAE)

The VAE was adapted to the continuous environment, since the VAE used for the Karel environment was designed for discrete tokens like `MOVE, IF, WHILE` [7]. For the continuous environment, the VAE is modified to handle tokens and parameters. Now it uses the same mechanism for calculating the *token_loss* (CrossEntropyLoss) mentioned in 6.2.1, but with an extra layer for *param_loss*. The *param_loss* uses Mean Squared Error (MSE) to compute the error in the predicted continuous parameters. To bound the value for `MOVE, TURN` tokens, a *sigmoid* and *tanh* mapping is added to ensure the VAE learns to map its distribution in a normalised range identical to the running agent.

7.1.6 PPO

For the custom continuous environment, the LEAPS framework is modified. It is divided into two parts. The VAE and Meta-Controller. The VAE is pretrained with a custom dataset of programs. The PPO algorithm trains the Meta-Controller, which takes an input program and outputs a latent vector. This latent vector is then decoded by the VAE and executed in the environment, and the return is calculated. The algorithm improves by analysing the return and updating the Meta-Controller network weights. The main purpose of the PPO algorithm is to teach the Meta-Controller to output a latent vector that generates an optimal sequence of actions to reach the goal and get maximum rewards. During training, the goal position is placed randomly at four corners of the environment as shown in Figures 7.4 and 7.6. The training trajectory demonstrating the reward mean and PPO loss is shown in 7.5. The main purpose of this step is set a start region for the CEM search to start with, by giving the best latent space with maximum return.

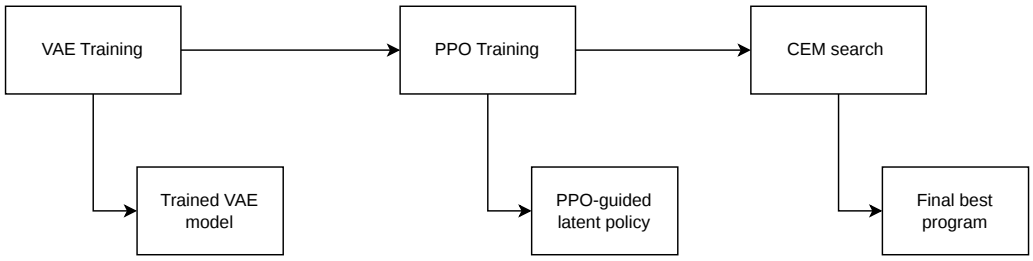


Figure 7.3: Current implementation of the modified LEAPS framework. Here, the VAE training stage takes the program dataset as input and produces a trained VAE model. The trained VAE decoder is then used during PPO training and CEM search to convert latent vectors into executable programs. The PPO trains on the environment and learns a policy (Meta-Controller) that generate useful program vectors. The trained PPO is used as a starting point for the CEM search. CEM search then searches the latent space around this initialization and outputs the best program

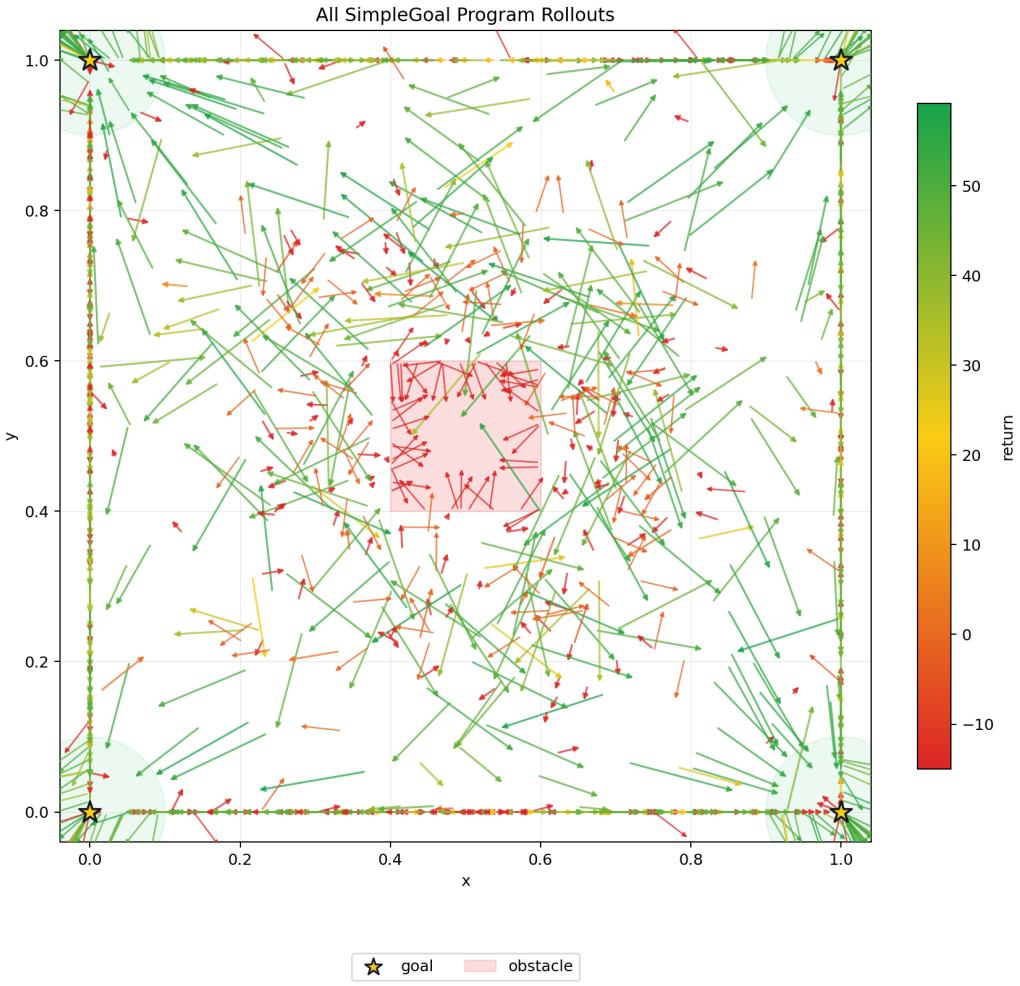


Figure 7.4: Quiver plot of PPO training with goal position placed randomly in four corners of the environment. The arrow indicates the final position and orientation of the agent. (NOTE: Some arrows are pointing away from the goal position because in that episode, the goal was placed in the other corners)

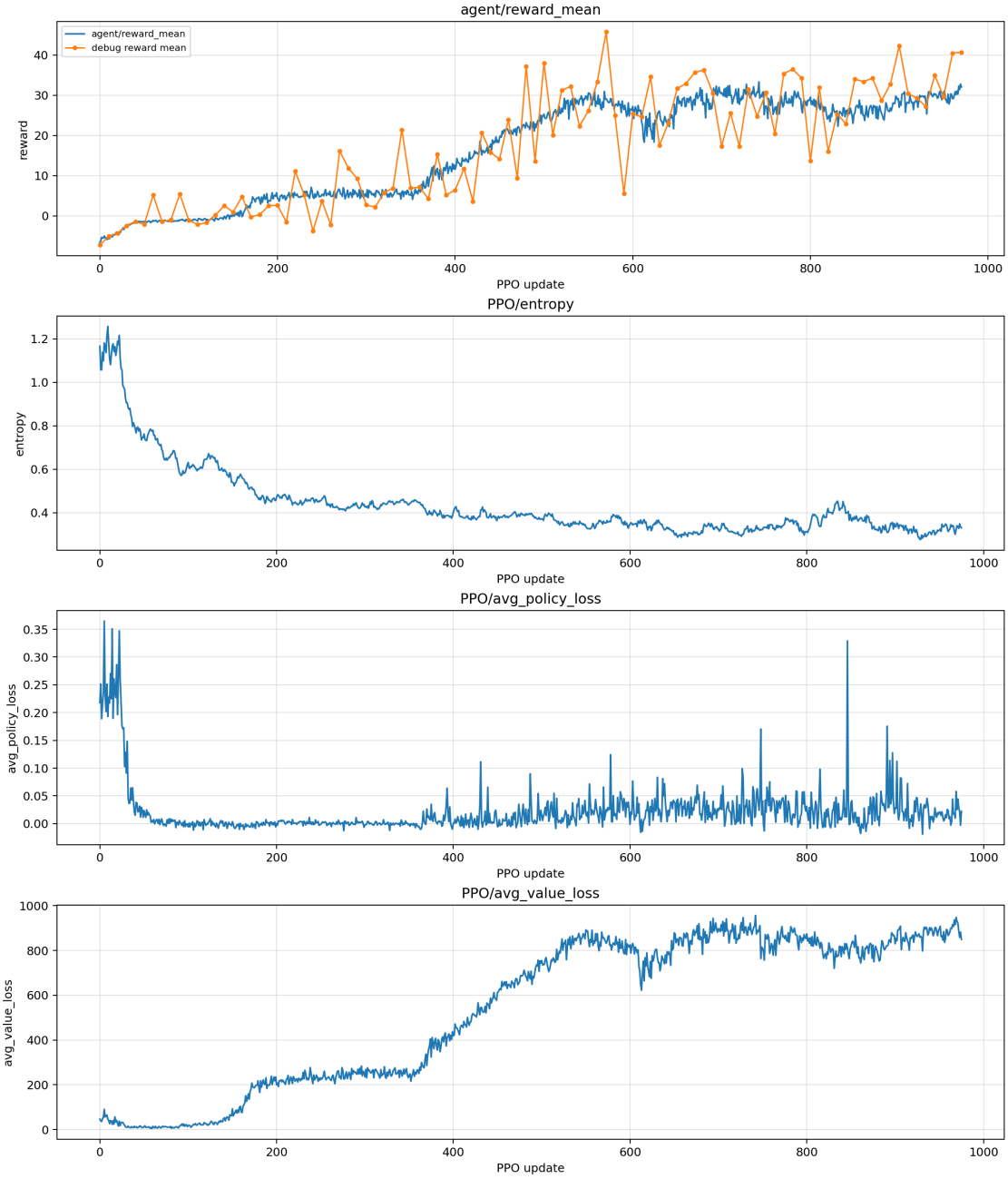


Figure 7.5: Plot shows the PPO training trajectory for the custom continuous environment. The policy learns to generate stable programs which is demonstrated by the increasing mean reward. The entropy decreases over time , it shows the transition from exploration to more deterministic policies. The policy loss stabilises after initial updates and occasional spike in policy loss is due to PPO being stochastic and the random goal position of the environment. The value loss increases as the critic gets adapted to the changing return

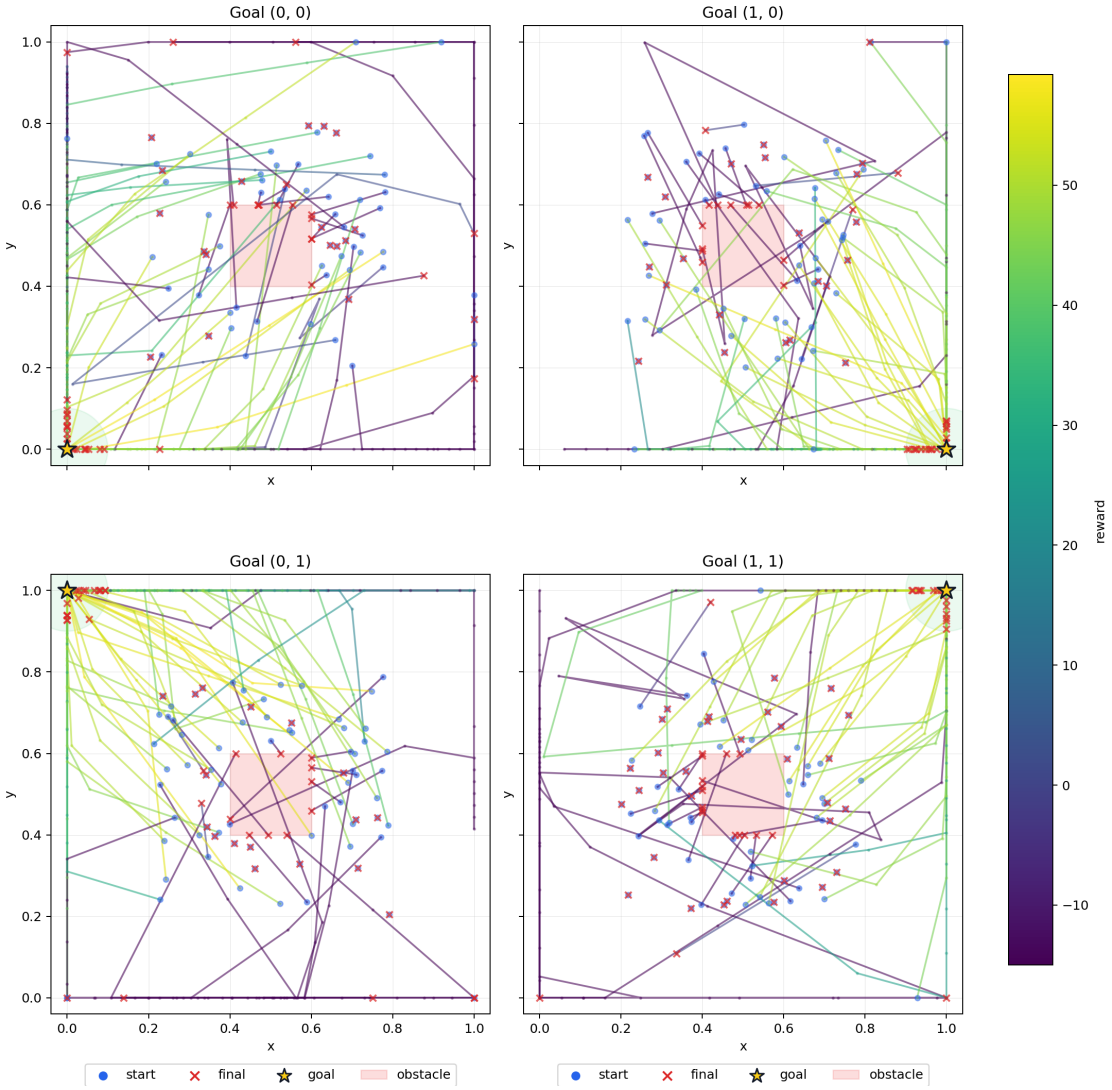


Figure 7.6: Plot shows the individual trajectory of the agent for each goal position. The trajectories are plotted by executing the program in the environment.

7.1.7 CEM Search

After the PPO training and pre-training of the VAE, the first stage is completed. The next stage is finding the best program which receives maximum return for each goal position. The CEM search is performed to find the best program for each goal position (goal placed at four corners of the environment). In this current implementation, the CEM search is given a starting point by the PPO training, which provides the best latent space with maximum return to start with. During the search, the agent's initial position, orientation, final position, and agent's trajectory are logged along with the best program and return. The best program and its respective trajectory are plotted on the environment map as shown below in the Figures7.7, 7.8, 7.9, and 7.10.

Agent Start Position = (0.482, 0.798) , Orientation = -2.051 rad
Agent Final Position = (0.051, 0.039), Orientation = -2.126 rad
Goal Position = (0.000, 0.000), Return = 58.79

Program :

```
MOVE 0.219

WHILE NOT NEAR_GOAL = x in [0.22,0.78] y in [0.22,0.78] DO {
  TURN -0.025
  MOVE 0.218
}

TURN -0.107
TURN -0.053

WHILE NOT NEAR_GOAL = x in [0.21,0.79] y in [0.21,0.79] DO {
  TURN -0.015
  MOVE 0.219
}

TURN -0.095
MOVE 0.200
TURN -0.099

WHILE NOT NEAR_GOAL = x in [0.22,0.78] y in [0.22,0.78] DO {
  TURN -0.011
  MOVE 0.217
}

TURN -0.103
TURN -0.057
MOVE 0.211
```

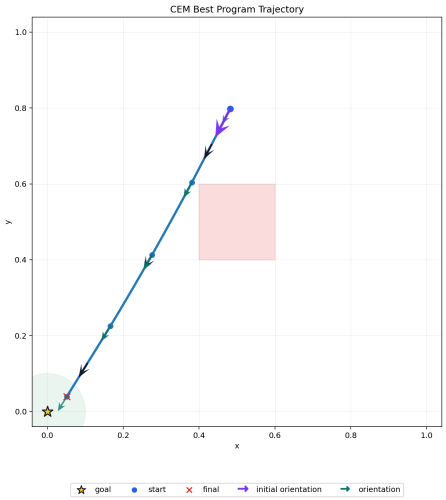


Figure 7.7: On the left side of this figure is the best program found by CEM search when the goal is positioned at *(0,0)*, and on the right side of this figure is the trajectory of the agent when the best program is executed in the environment.

Agent Start Position = (0.778, 0.248) , Orientation = 1.593 rad
Agent Final Position = (0.021, 0.967), Orientation = 2.717 rad
Goal Position = (0.000, 1.000), Return = 60.40

Program :

```
WHILE NOT FACING_GOAL 0.825 rad DO {
  TURN 0.309
}

WHILE NOT NEAR_GOAL = x in [0.20,0.80] y in
[0.20,0.80] DO {
  TURN 0.281
  MOVE 0.220
}

MOVE 0.222
WHILE NOT FACING_GOAL 0.816 rad DO {
  TURN 0.277
}

MOVE 0.217
MOVE 0.215

WHILE NOT NEAR_GOAL = x in [0.20,0.80] y in
[0.20,0.80] DO {
  TURN 0.179
  MOVE 0.217
}

MOVE 0.215
```

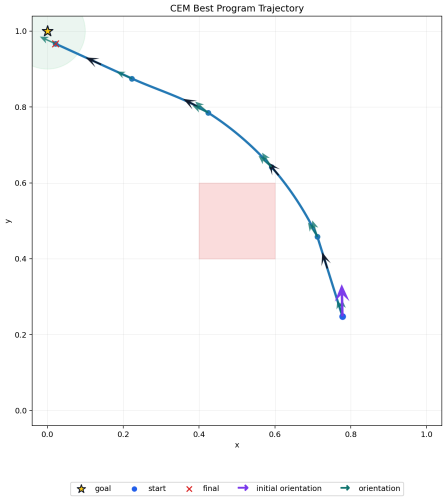


Figure 7.8: On the left side of this figure is the best program found by CEM search when the goal is positioned at *(0,1)*, and on the right side of this figure is the trajectory of the agent when the best program is executed in the environment.

Agent Start Position = (0.242, 0.541) , Orientation = -0.776 rad
Agent Final Position = (0.986, 0.002), Orientation = -0.578 rad
Goal Position = (1.000, 0.000) Return = 58.57

Program :

```
MOVE 0.227
TURN 0.198
MOVE 0.233

WHILE NOT FACING_GOAL 0.863 rad DO {
  TURN 0.394
}

WHILE FACING_GOAL 0.851 rad DO {
  MOVE 0.231
}

WHILE FACING_GOAL 0.829 rad DO {
  MOVE 0.230
}

WHILE NOT FACING_GOAL 0.852 rad DO {
  TURN 0.256
  MOVE 0.234
}

WHILE FACING_GOAL 0.844 rad DO {
  MOVE 0.229
}
```

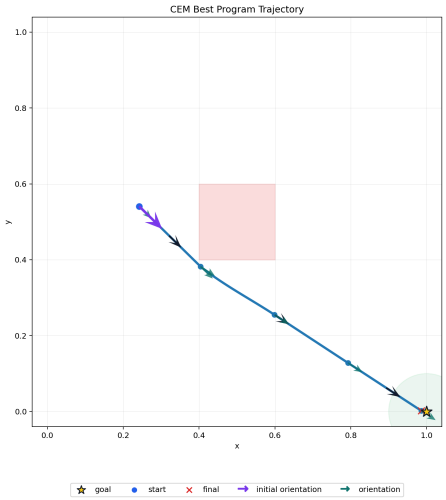


Figure 7.9: On the left side of this figure is the best program found by CEM search when the goal is positioned at *(1,0)*, and on the right side of this figure is the trajectory of the agent when the best program is executed in the environment.

Agent Start Position = (0.203, 0.353) , Orientation = 1.211 rad
Agent Final Position = (0.961, 1.000), Orientation = 0.381 rad
Goal Position = (1.000, 1.000), Return = 59.64

Program :

```
WHILE NEAR_OBSTACLE = x in [0.30,0.70] y in [0.30,0.70] DO {
  MOVE 0.208
  WHILE NOT FACING_GOAL 0.785 rad DO {
    TURN 0.025
  }
}

WHILE NOT FACING_GOAL 0.775 rad DO {
  TURN -0.141
}

WHILE NOT NEAR_OBSTACLE = x in [0.30,0.70] y in [0.30,0.70] DO
{
  TURN -0.166
  MOVE 0.206
}

WHILE NOT FACING_GOAL 0.765 rad DO {
  TURN -0.169
}
```

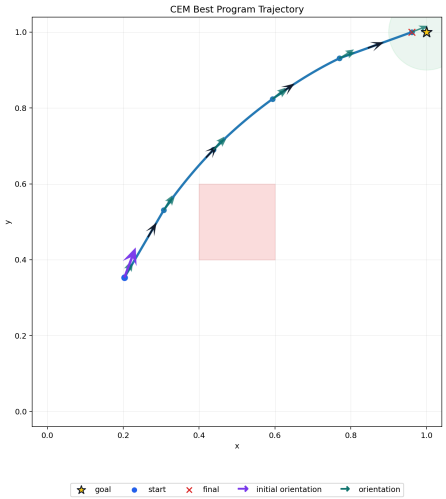


Figure 7.10: On the left side of this figure is the best program found by CEM search when the goal is positioned at *(1,1)*, and on the right side of this figure is the trajectory of the agent when the best program is executed in the environment.

7.1.8 Results

Through this thesis, we successfully modified the LEAPS framework to work in a continuous environment. The environment is similar to the four corners environment in Karel [7]. The potential of this experiment is the application of explainable reinforcement learning in real-world applications, where the end user can tweak the program to improve the performance of the agent. Moreover, the agent not only accomplished reaching the goal, but it also used control statements to effectively navigate the environment, which reduced the total number of tokens used.

The usability of this approach will eventually contribute to solving the scarcity of system controllers to some extent, as people with minimal experience and basic programming knowledge can tweak and optimise system controllers effectively.

In Karel, the agent navigates using discrete actions, which won't be efficient in reaching the goal position. For instance, in the case of the *FourCorner* Karel environment as shown in the figure 7.11, if the agent and goal position are placed at opposite corners, the diagonal navigation will be the most efficient and shortest one (a straight line is the shortest distance between two points), but the existing LEAPS framework followed a discrete action space because of which the agent cannot move diagonally [32]. In our approach, with the goal and agent placed diagonally opposite to each other, the agent efficiently manoeuvres to the goal position without hitting the obstacle, as shown in the figure 7.12, since we used a continuous action space.

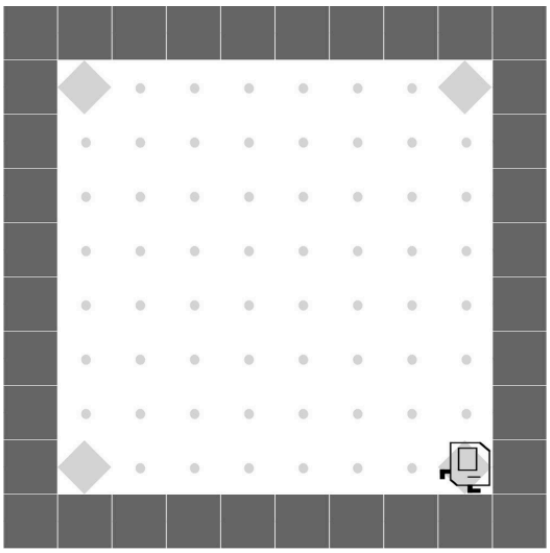


Figure 7.11: Fourcorner Karel tasks [7]

Agent Start Position = (0.000, 0.000) , Orientation = 0.109 rad
Agent Final Position = (1.000, 0.965), Orientation = 1.239 rad
Goal Position = (1.000, 1.000), Return = 63.36

Program :

```
MOVE 0.214

WHILE NOT NEAR_GOAL = x in [0.20,0.80] y in [0.20,0.80] DO {
  TURN 0.226
  MOVE 0.219
}

MOVE 0.218

IF NOT NEAR_BORDER = edge < 0.18 {
  MOVE 0.216
}

TURN 0.048
MOVE 0.215

WHILE NOT FACING_GOAL 0.811 rad DO {
  TURN 0.186
}

MOVE 0.216
TURN 0.168

WHILE NOT NEAR_GOAL = x in [0.20,0.80] y in [0.20,0.80] DO {
  TURN 0.229
  MOVE 0.220
}
```

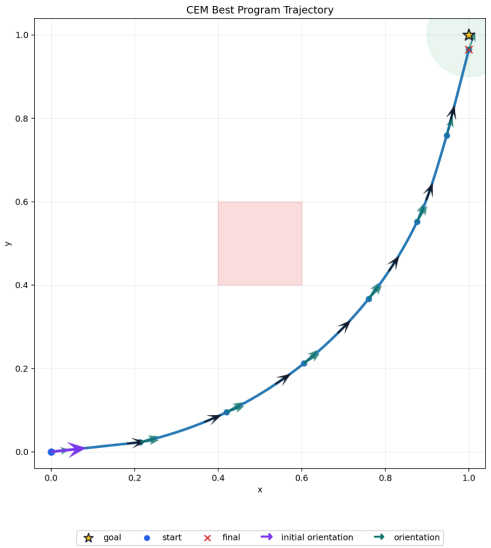


Figure 7.12: Agent navigating from one corner to the opposite goal position efficiently

From the figure 7.12, the program on the left side shows the explanation of the agent's action, mainly it answers the questions "Why?", "How?", for instance, we can easily interpret why the agent took the action `TURN 0.226`, it's because its current position was not near the goal.

Chapter 8

Validation

To validate the performance of this framework in terms of loss and return, a systematic experiment is conducted. Through this experiment we validate the consistency of the final CEM search. The experiment comprises 5 VAEs, each with 5 PPOs and each PPO with 10 CEM searches, resulting in 250 CEM searches in total. The main objective of this validation is to check the framework's stability. Since VAE and PPO are all neural networks, they can differ in the initial weights and biases, so to verify that the CEM search finds programs with the highest return regardless of VAE and PPO initialisation, this experiment is conducted. The results of the experiment are presented below

VAE	Mean Token Loss	Mean Param. MSE	Mean CEM Return	Best CEM Return
vae_1	0.1898	0.3457	58.454±1.473	62.930
vae_2	0.1825	0.3452	58.278±0.843	59.867
vae_3	0.1792	0.3450	58.232±0.929	60.981
vae_4	0.1819	0.3444	58.452±1.180	62.110
vae_5	0.1813	0.3451	58.478±0.943	60.667

Table 8.1: VAE reconstruction losses and CEM search performance. Each VAE has 50 CEM runs (5 PPO and 10 CEM searches). The CEM mean return ± standard deviation are computed over the best return obtained in each CEM run for the corresponding VAE.

Figure 8.1 shows 5 independent VAE runs. The total loss, token loss, parameter loss, and KL loss all show nearly identical behaviour for all VAEs. It shows that the VAE training is stable, and for repeated runs, similar results are demonstrated.

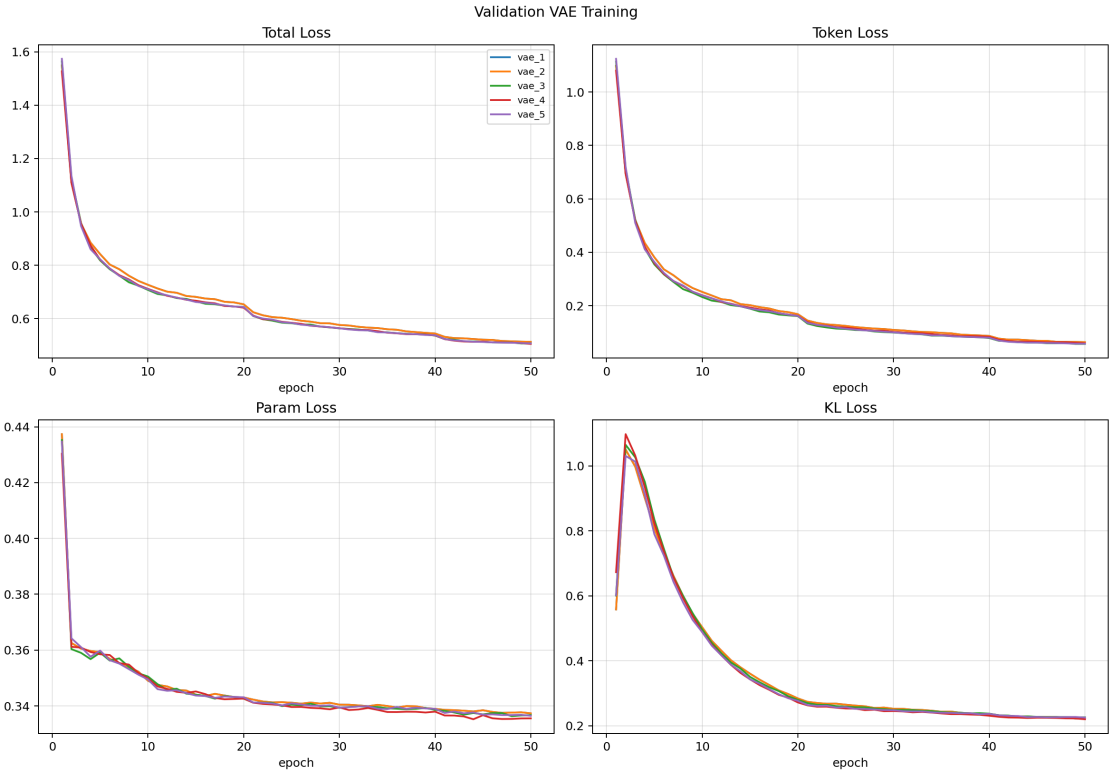


Figure 8.1: 5 VAE training runs

Figure 8.2, 8.3 shows the 250 CEM searches. All the CEM search plateaus around 60 total rewards. This shows the CEM search stability. Even though we used random goal positions in these experiments, the CEM search consistently found the best program across all 250 searches.

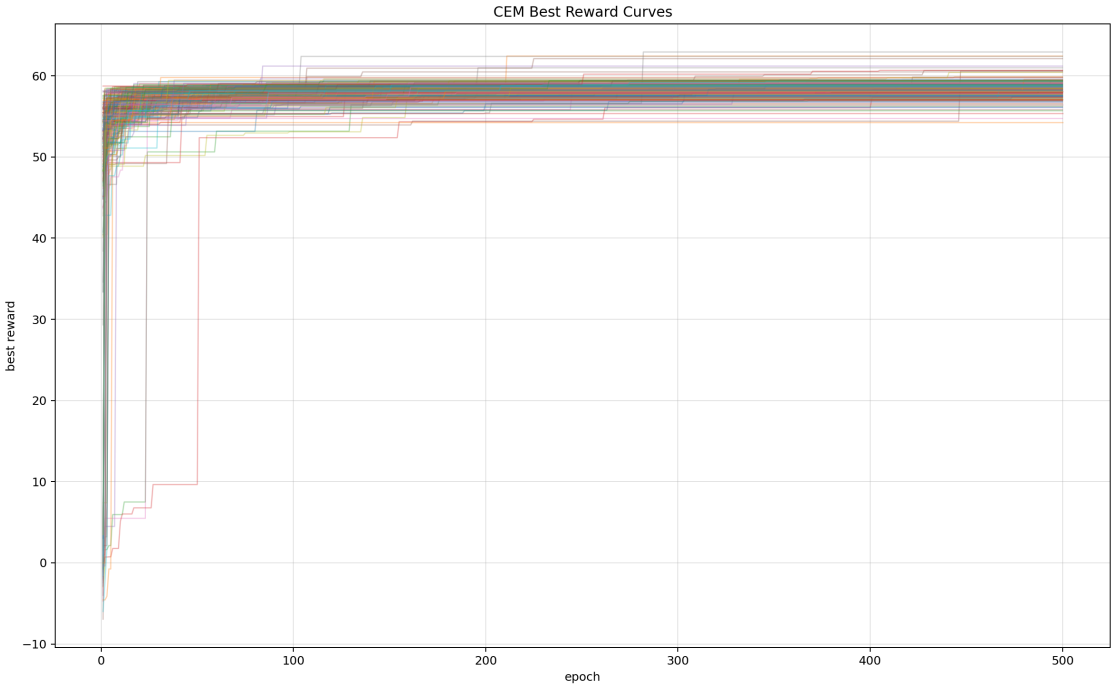


Figure 8.2: 250 CEM search runs

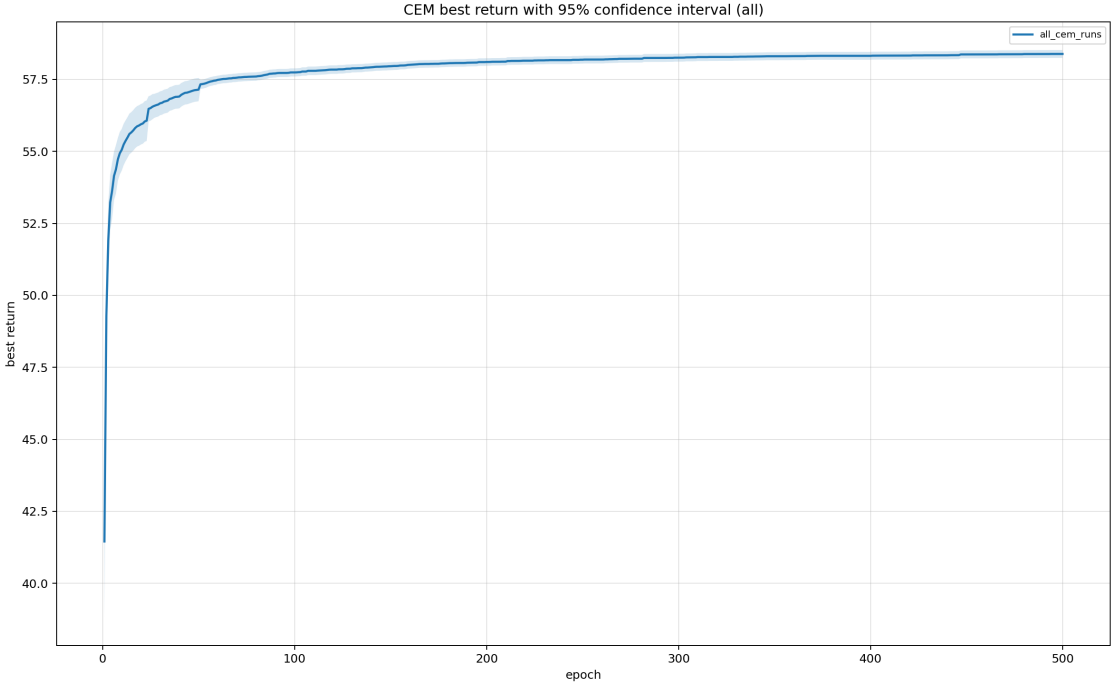


Figure 8.3: CEM best return mean with 95% confidence interval for 250 CEM search runs

This validation experiment reinforces the framework's ability to generate programs with maximum returns for dynamic goal positions. The Figure 8.4 shows a sample trajectory plot depicting all four goal positions and the respective agents' start positions

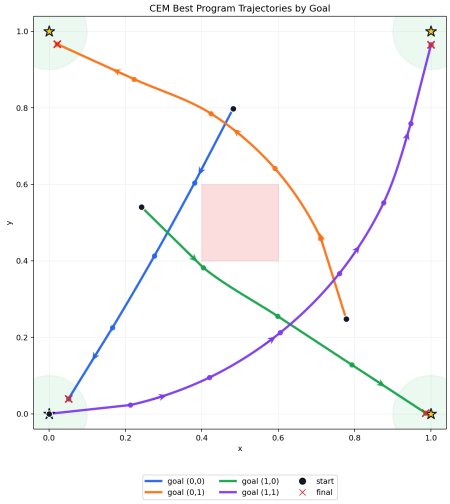


Figure 8.4: Sample CEM search plot showing all four goal positions and the corresponding agent's starting positions

Chapter 9

Discussion

In the validation experiment, the CEM search converged to high returns in almost all runs. This shows that in this framework, the VAE's latent space quality is important. The VAE training results support this. In all independent runs, the VAEs were demonstrating near-identical loss curves, which showed that the latent space was stable and reproducible.

The token loss converges to near-zero rapidly, while the parameter loss converges more slowly. This shows that learning continuous parameter distributions introduces additional complexity than a discrete token sequence.

One interesting observation from the generated programs is the extensive use of "`WHILE NOT NEAR_GOAL`" loops, rather than using flat "`IF-ELSE`". The CEM search found that loop-based navigation is token efficient. This was achieved by a combination of reward shaping mechanisms. The DSL was designed in a minimal way to balance expressiveness and explainability. If the vocabulary is increased, complex behaviour can be expressed, but the understanding of the generated programs and training the VAE becomes difficult. However, extending the DSL vocabulary is necessary in solving complex problems in future.

The continuous action space demonstrated a clear advantage over the original discrete LEAPS framework. In the FourCorner task, the agent was allowed to move only in grid-aligned movements, restricting it to navigate in a diagonal direction. In contrast, the agent can reach the goal in a shorter path in a continuous environment.

The 16-channel CNN-compatible observation encoding was an adaptation from the original LEAPS framework. In this, to encode orientation instead of using raw angle values, $\cos(\theta)$ and $\sin(\theta)$ are used; this solved the $\pm\pi$ discontinuity problem while encoding. This design choice supported the stable and consistent training as seen in all VAE runs.

The PPO training is now used as a start point for the CEM search, and the VAE is used as the decoder of the program. The PPO can be used as a decoder. Although the generated programs received maximum return, it was missing the proper structure, which affected the readability and needed post-processing to make them user-friendly. For the tokens, the VAE directly supervises the `MOVE` and `TURN` tokens parameters. The condition-token parameters are not directly supervised by the VAE parameter loss in the current implementation. Instead, they are handled with hardcoded threshold values, since the goal positions, obstacle positions, and boundaries are known in advance. For a more dynamic unseen environment, the conditional tokens parameter should also be supervised by the VAE parameter loss, allowing the model to learn environment-adaptive thresholds.

Chapter 10

Future Work

As the field of XRL expands in all dimensions, the application of programmatic policy synthesis is getting traction. The demand for XRL is increasing since it increases transparency and trust through explainability. This gives humans the answers to the questions "Why?" and "How?". Explainability in RL is crucial because, in real-world applications like robotics and autonomous driving, a model learning from experience is preferred. Since in these scenarios, getting a static dataset to train a model for all situations is nearly impossible. This thesis work contributes as a basic step in implementing programmatic explanation of RL in continuous environments, useful for real-world applications. This work was conducted in a controlled environment with limited complexity and environment-specific DSL, VAE and CEM search. As a next step, the complexity of the environment can be increased, for instance, instead of a fixed obstacle, the environment can be extended to multiple and dynamic obstacles, and the DSL, VAE and CEM search can be made dynamic. Instead of a navigation task, this approach can also be extended to set-point based control problems where, given the description of a scenario in the state space, the agent has to provide a programmatic statement to reach the desired operation region. Another interesting research direction in this domain is to implement this approach in more high dimensional, high dynamic environments like robotic arms or autonomous driving, where the explainability of the policy is critical and a clear goals or set-points are not straightforward.

In real-world applications, each problem statement can be converted into a multi-objective problem; for instance, travelling from A to B can be considered as the first objective, and completing this travel in a short time can be considered as the second objective. Similarly, this research can be extended to multi-objective reinforcement learning, which aligns closely with real-life scenarios. The environment used in this research can be extended to a multi-objective one by considering the velocity of the agent while navigating. Giving a human-guided importance factor to each objective can make the training of these agents easier and align this approach closer to real-world decision-making.

These scopes of implementation show that the usage of XRL is not limited; it can be extended to any domain with proper design and training, and has the potential to make the black-box nature of complex models a part of history, not the future.

Chapter 11

Conclusion

In this thesis, we present the extended version of LEAPS capable of operating in a continuous control environment. This work addresses the primary limitation of the LEAPS framework, i.e., confined to a discrete action space. The key contributions of this work are as follows.

First, we designed a custom continuous Gymnasium environment and a DSL capable of expressing and explaining continuous navigation behaviour. This involved setting parametrised `MOVE`, `TURN` tokens and threshold-bounded conditional predicates.

Second, we modified the VAE to handle discrete tokens and continuous parameters. This was demonstrated in multiple runs, showing consistent training performance.

Third, we showed that the CEM search was able to find interpretable programmatic policies from the learned latent space. The main advantage of generated programs is their direct explainability. Each action is accompanied by the condition which triggered it, and thereby, we can understand why the agent took that action in that particular step, answering the "Why" question and explaining the "How" question clearly.

This approach is beneficial in safety-critical domains like healthcare and autonomous systems, where accountability is an important factor, because the decision-making process can be explained in a transparent way.

Overall, this work demonstrated its applicability in a continuous environment and programmatic explainability of RL agents. This contributes as a step towards the development of explainable, accountable control systems, especially in real-world applications.

Bibliography

[1] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997, ISBN: 9780071154673.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction* (Adaptive computation and machine learning series), en, Second edition. Cambridge, Massachusetts: The MIT Press, 2018, ISBN: 978-0-262-03924-6.

[3] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[4] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, Feb. 2015. DOI: 10.1038/nature14236.

[5] D. Gunning and D. W. Aha, "Darpa's explainable artificial intelligence program," *AI Magazine*, vol. 40, no. 2, pp. 44–58, 2019. DOI: https://doi.org/10.1609/aimag.v40i2.2850. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1609/aimag.v40i2.2850. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v40i2.2850.

[6] H. Block, "A review of "perceptrons: An introduction to computational geometry," *Information and Control*, vol. 17, no. 5, pp. 501–522, 1970, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(70)90409-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995870904092.

[7] D. Trivedi, J. Zhang, S.-H. Sun, and J. J. Lim, "Learning to synthesize programs as interpretable and generalizable policies," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=wP9twkexC3V.

[8] A. B. Arrieta et al., *Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai*, 2019. arXiv: 1910.10045 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1910.10045.

[9] Y. Bekkemoen, "Explainable reinforcement learning (xrl): A systematic literature review and taxonomy," *Machine Learning*, vol. 113, pp. 1–87, Nov. 2023. DOI: 10.1007/s10994-023-06479-7.

[10] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/e6d8545daa42d5ced125a4bf747b3688-Paper.pdf.

[11] J. P. Inala, O. Bastani, Z. Tavares, and A. Solar-Lezama, "Synthesizing programmatic policies that inductively generalize," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020. [Online]. Available: https://openreview.net/forum?id=S1l8oANFDH.

[12] M. Ghasemi and D. Ebrahimi, "Introduction to reinforcement learning," *arXiv preprint arXiv:2408.07712*, 2024.

[13] J. Terven, "Deep reinforcement learning: A chronological overview and methods," *AI*, vol. 6, p. 46, Feb. 2025. DOI: 10.3390/ai6030046.

[14] L. P. Kaelbling, M. L. Littman, and A. W. Moore, *Reinforcement learning: A survey*, 1996. arXiv: cs/9605103 [cs.AI]. [Online]. Available: https://arxiv.org/abs/cs/9605103.

[15] Y. LeCun and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, May 2015. DOI: 10.1038/nature14539.

[16] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, Apr. 2014. DOI: 10.1016/j.neunet.2014.09.003.

[17] M. Sewak, *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*. Jan. 2019, ISBN: 978-981-13-8284-0. DOI: 10.1007/978-981-13-8285-7.

[18] V. Mnih et al., *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1312.5602.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1707.06347.

[20] V. Mnih et al., *Asynchronous methods for deep reinforcement learning*, 2016. arXiv: 1602.01783 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1602.01783.

[21] F. Sovrano, F. Vitali, and M. Palmirani, "Making things explainable vs explaining: Requirements and challenges under the gdpr," in *AI Approaches to the Complexity of Legal Systems XI-XII*. Springer International Publishing, 2021, pp. 169–182, ISBN: 9783030898113. DOI: 10.1007/978-3-030-89811-3_12. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-89811-3_12.

[22] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing*, vol. 73, pp. 1–15, Feb. 2018, ISSN: 1051-2004. DOI: 10.1016/j.dsp.2017.10.011. [Online]. Available: http://dx.doi.org/10.1016/j.dsp.2017.10.011.

[23] A. Fernandez, F. Herrera, O. Cordon, M. Jose del Jesus, and F. Marcelloni, "Evolutionary fuzzy systems for explainable artificial intelligence: Why, when, what for, and where to?" *Comp. Intell. Mag.*, vol. 14, no. 1, pp. 69–81, Feb. 2019, ISSN: 1556-603X. DOI: 10.1109/MCI.2018.2881645. [Online]. Available: https://doi.org/10.1109/MCI.2018.2881645.

[24] M. Gleicher, "A framework for considering comprehensibility in modeling," *Big Data*, vol. 4, no. 2, pp. 75–88, 2016. DOI: 10.1089/big.2016.0007. [Online]. Available: https://doi.org/10.1089/big.2016.0007.

[25] M. W. Craven and J. W. Shavlik, "Extracting comprehensible models from trained neural networks," 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:121360.

[26] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti, *A survey of methods for explaining black box models*, 2018. arXiv: 1802.01933 [cs.CY]. [Online]. Available: https://arxiv.org/abs/1802.01933.

[27] Z. C. Lipton, *The mythos of model interpretability*, 2017. arXiv: 1606.03490 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1606.03490.

[28] D. Gunning, E. Vorm, J. Wang, and M. Turek, "Darpa 's explainable ai (xai) program: A retrospective," *Applied AI Letters*, vol. 2, Dec. 2021. DOI: 10.1002/ail2.61.

[29] A. M. Salih et al., "A perspective on explainable artificial intelligence methods: Shap and lime," *Advanced Intelligent Systems*, vol. 7, no. 1, 2024, ISSN: 2640-4567. DOI: 10.1002/aisy.202400304. [Online]. Available: http://dx.doi.org/10.1002/aisy.202400304.

[30] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, USA, 2016, pp. 1135–1144.

[31] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, *Programmatically interpretable reinforcement learning*, 2019. arXiv: 1804.02477 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1804.02477.

[32] I. Higgins et al., "Beta-VAE: Learning basic visual concepts with a constrained variational framework," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=Sy2fzU9gl.