Yu He

Combining Classical Algorithms and Deep Reinforcement Learning Agents

Computer Science Tripos – Part II

Murray Edwards College

May 13, 2022

Declaration

I, Yu He of Murray Edwards College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Yu He*

Date *13/05/2022*

Acknowledgements

I want to express my immense gratitude to my supervisors, Andreea-Ioana Deac and Prof. Pietro Liò, for their generous guidance and encouragement throughout the project. I am also sincerely grateful to Dr Petar Veličković for introducing me to the project and for his invaluable advice. In addition, I am thankful to Han Xuanyuan and Jacky Kung for their very helpful comments on this dissertation. Lastly, I would like to thank my Director of Studies, Luana Bulat, for kindly supporting me over the last three years.

Proforma

Name:	**2428G**
Project Title:	**Combining Classical Algorithms and Deep Reinforcement Learning Agents**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**11931**[1]
Line Count:	**3761**[2]
Project Originator:	Andreea-Ioana Deac
Supervisor:	Andreea-Ioana Deac, Prof. Pietro Liò

Original Aims of the Project

This project approaches reinforcement learning problems by combining classical algorithm subroutines into intelligent agents. It looks at a recent proposal, eXecuted Latent Value Iteration Net (XLVIN), which exploited the benefits of simulating a dynamic programming algorithm within a deep reinforcement learning agent for better decision-making strategies. This project aims to implement and evaluate XLVIN, and then build upon it to address its limitations, extending its application scope to handle a range of intrinsically more complex reinforcement learning problems.

Work Completed

The project has completed all success criteria and extensions. I successfully trained a graph neural network to simulate a value iteration algorithm. Then, I used it as a core processor to construct the XLVIN model and compared it against a state-of-the-art agent, demonstrating its effectiveness on *discrete* control problems. Moreover, I identified the architectural bottlenecks suffered by XLVIN and then proposed a novel design named **Continuous Neural Algorithmic Planner (CNAP)** to handle *continuous* control problems with highly *complex* dynamics. CNAP is capable of a much broader application scope closer to real-life domains while inheriting the sample efficiency from XLVIN.

Special Difficulties

None.

[1] This word count was computed using `texcount` (`https://app.uio.no/ifi/texcount/`), with `%TC:group tabular 1 1` and `%TC:group table 0 1` to include tables.

[2] This line count was computed using `cloc` (`https://github.com/AlDanial/cloc`).

Contents

Chapter 1

Introduction

This dissertation explores the benefits of endowing classical algorithm subroutines into Deep Reinforcement Learning agents. It presents Continuous Neural Algorithmic Planners (CNAPs), extending from a recent proposal by addressing its architectural bottlenecks with novel solutions. CNAPs expanded the original framework's application scope from simple discrete control to complex continuous control tasks. We will see that CNAPs can demonstrate outstanding performances against a state-of-the-art model across a range of Reinforcement Learning tasks.

1.1 Motivation

Reinforcement Learning (RL) considers a paradigm of problems involving a goal-oriented agent that learns through interactions. It mimics a close form of learning that humans do, and is considered to be a stepping stone towards the general principles of intelligence [1].

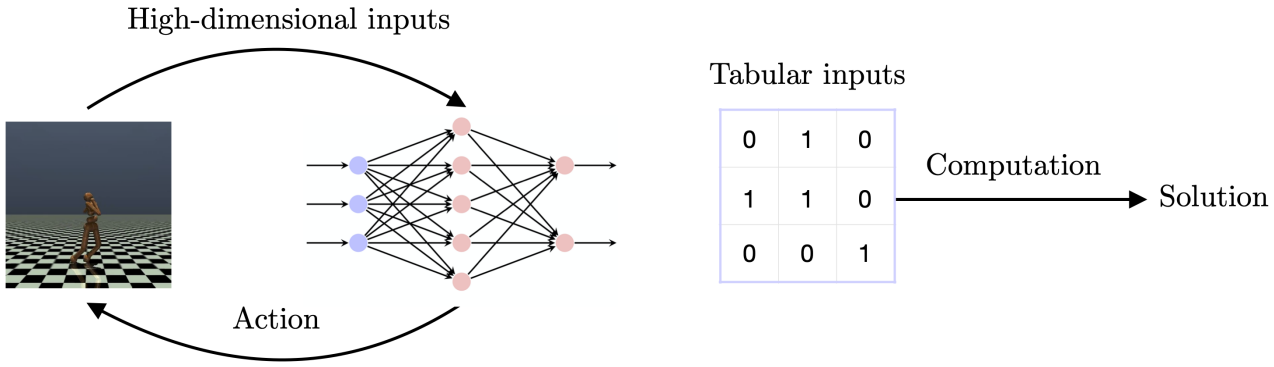


Figure 1.1.1: (a) Deep Reinforcement Learning and (b) Classical Algorithm

Leveraging benefits from deep neural networks, *Deep Reinforcement Learning (Deep RL)* [2] (Figure 1.1.1a) is a powerful approach to learning directly from high-dimensional unstructured input data. It has demonstrated a wide range of applications, such as autonomous driving [3], dialogue generation [4] and healthcare treatment [5]. However, deep learning is known for being data-hungry, requiring many samples to train the large set of model parameters. The problem is exacerbated by the complexity of RL problems. Current agents such as DeepMind's MuZero [6] can require 10-50 years of experience per game, making them impractical to real-world tasks.

On the other hand, *classical algorithms* (Figure 1.1.1b) target the heart of the problem itself, revealing the underlying reasoning of the problem-solving process. Algorithms intrinsically generalise and provide mathematical provability and performance guarantees. However, algorithms often impose rigid assumptions on the input format, inhibiting them from solving more complex problems.

An emerging field, *Neural Algorithmic Reasoning* [7], looks at learning classical algorithms with deep neural networks to achieve the best of both worlds [8][9][10]. Deep learning elevates the input constraints of algorithms by representing natural data as feature vectors in high-dimensional space. Conversely, algorithms instil proper inductive bias into the neural network to guide the learning with better sample efficiency and generalisation ability. The mutually beneficial relationship sets the ground for this project, where it introduces classical algorithmic reasoning into Deep RL to bring about a highly scalable agent for complex control tasks.

1.2 Previous Work

Following the direction of combining classical algorithms and Deep RL, this project looks at a very recent work, **eXecuted Latent Value Iteration Nets (XLVINs)**, presented by Deac et al. [11] at the 35^{th} Conference on Neural Information Processing Systems (NeurIPS 2021).

XLVINs' central idea is to align *Graph Neural Networks (GNNs)* with *value iteration* to imitate the algorithm's behaviour. Value iteration is a classical dynamic programming algorithm that guarantees to solve an RL problem, but it requires a tabular representation of the environment. The usage of a GNN to simulate value iteration enables the handling of high-dimensional raw data directly. On the other hand, value iteration gives the GNN a reasoning power to analyse the RL problem and produce better decision-making strategies. Together, XLVIN agents can learn to select actions intelligently within a low data regime.

XLVINs have demonstrated successful applications in some environments, such as grid-world mazes, classic control, and pixel-based Atari. However, these environments only belong to a subset of RL problems due to the following limitations that constrain XLVINs.

- **Discrete control:** XLVIN agents can only choose an action from a *discrete* set, such as pushing left or right, but not from a *continuous* range, such as pushing with a specific magnitude in the range of $[0, 1]$.
- **Singular dimensional control:** XLVINs can only handle a *single* action at a time, such as turning one particular joint. An action *vector* allows a higher degree of control, such as instructing a robotic dog to run by operating its four legs simultaneously.
- **Relatively simple dynamics:** XLVINs are also restricted by a *small* action space, only allowing a limited set of action choices. This restriction is due to an architectural bottleneck which will be discussed in later chapters.

Continuous action space adds extra difficulty to the decision-making agent due to the infinite pool of action choices, requiring more careful treatments and thinking. Nevertheless, continuous control is of significant importance, as most simulation or robotics control tasks [12] have continuous action spaces by design. Furthermore, high complexity naturally arises as the problem moves towards more powerful real-world domains, such as DeepMind's recent application in controlling nuclear fusion plasma with Deep RL [13]. The scalability of how well the agents can handle *highly complex continuous control* under rich contact with the environment is substantial in filling the gap between a theoretical framework and its physical embodiment.

1.3 Contribution of my project

My project was a success, meeting all Success Criteria and Extensions. I first implemented the XLVIN model and evaluated its performance on *discrete* control problems. Based on XLVINs, I then proposed a novel architecture **Continuous Neural Algorithmic Planners (CNAPs)**, which can tackle *continuous* action space problems. Most notably, CNAPs can scale up to high-dimensional continuous control problems with *complex* dynamics. The successful expansion in application scope demonstrates that such a Deep RL agent with algorithmic reasoning power can be applied to domains with more real-world interests, such as autonomous driving, virtual environments, and robotics simulation.

Furthermore, the contribution of my project is not limited to Deep RL. Also as a subject of interest to Neural Algorithmic Reasoning, my project introduces a novel set-up to align *discrete* GNN processes with *discrete* algorithms under *continuous* inputs. The results based on this project were awarded **Best Paper Finalist** at GroundedML: Workshop on Anchoring Machine Learning in Classical Algorithmic Theory, the 10^{th} International Conference on Learning Representations (ICLR 2022) [14]. I would also like to further explore the topic in future, given its exciting potential.

1.4 Structure of dissertation

The dissertation is structured as follows: Chapter 2 presents the theoretical background required to understand this project, as well as the undertaken software engineering approach. Chapter 3 explains the composition of XLVINs and CNAPs and the algorithms involved, including the construction of the training and testing pipelines. Then in Chapter 4, I present the quantitative and qualitative analyses performed to demonstrate the effectiveness of both XLVINs and CNAPs. Finally, I discuss the achievement and reflection in Chapter 5 and point out possible future work directions.

Chapter 2

Preparation

In this chapter, I introduce the background theory (2.1) required to understand the project, followed by a documentation on the starting point (2.2) and requirement analysis (2.3). Then I describe the software engineering techniques (2.4) used throughout the execution of this project.

2.1 Background Theory

This section starts by introducing Reinforcement Learning (2.1.1) as the context of this project. Then it describes Graph Neural Networks (2.1.2) with an explanation of their basics and how they act as the core for Neural Algorithmic Reasoning (2.1.3). This leads to a discussion on how classical algorithm subroutines can be introduced into Reinforcement Learning agents. Finally, it introduces Proximal Policy Optimisation (2.1.4) as the training algorithm for the models.

2.1.1 Reinforcement Learning

A Reinforcement Learning (RL) problem involves an *agent* that observes the *state* of the environment and learns to take *actions* in order to achieve a *goal*. It can be formally described using the Markov Decision Process framework.

Markov Decision Process (MDP):

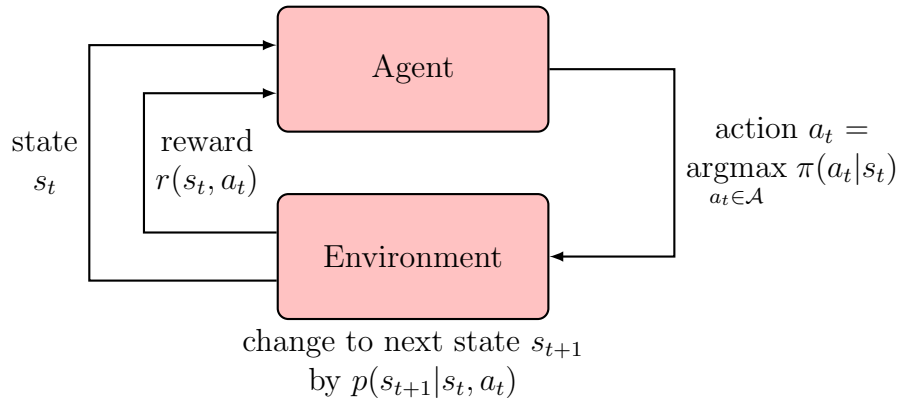


Figure 2.1.1: Framework of Reinforcement Learning

As illustrated in Figure 2.1.1, at each timestep $t \in \{0, 1, ..., T\}$, the agent performs an action $a_t \in \mathcal{A}$ under the current state $s_t \in \mathcal{S}$ according to a policy π. A policy $\pi(a_t|s_t)$ defines the probability of selecting action a_t given state s_t. The action spawns a transition from state s_t into a new state s_{t+1} by a transition probability $p(s_{t+1}|s_t, a_t)$, and produces an immediate reward $r_t = r(s_t, a_t)$. Transition and reward functions assume Markov property of memorylessness, so they depend only on the current state following an action. The cumulative rewards received over time can be specified as the infinite horizon discounted return:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.1.1}$$

where trajectory $\tau = ((s_0, a_0), (s_1, a_1), (s_2, a_2), ...)$ is the sequence of state-action pair the agent has experienced, and $\gamma \in [0, 1]$ is the discount factor that determines how much emphasis is put in the distant future rather than the immediate future.

The goal of the agent is to maximise the above return it receives in the entire episode. In summary, an RL problem can be stated as finding the *optimal policy* which maximises the overall return:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim\pi}[R(\tau)] \tag{2.1.2}$$

where the trajectory τ is generated according to the policy π. And at timestep t, we can evaluate how desirable a state s is in the long term by the *state-value function*:

$$V^*(s) = \mathbb{E}_{\tau\sim\pi^*}[R(\tau)|s_t = s] \tag{2.1.3}$$

Value Iteration algorithm:

Value Iteration (VI) [15] is a *classical dynamic programming algorithm* that computes the optimal policy π^* and its state-value function $V^*(s)$ given a fully known MDP. It randomly initialises $V^*(s)$ and iteratively updates it using the Bellman optimality equation [15]:

$$V^*_{i+1}(s) = \max_{a\in\mathcal{A}}\{r(s, a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s, a)V^*_i(s')\} \tag{2.1.4}$$

The optimal state-value function $V^*(s)$ is found at convergence. We can then extract the optimal policy π^* by finding which action leads to the optimal state-value function:

$$\pi^*(s) = \underset{a\in\mathcal{A}}{\operatorname{argmax}}\{r(s, a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s, a)V^*_i(s')\} \tag{2.1.5}$$

Algorithm 1 Value Iteration

Randomly initialise $V_0(s)$ for all $s \in \mathcal{S}$
repeat
  for $s \in \mathcal{S}$ **do**
    $V^*_{i+1}(s) \leftarrow \text{BellmanUpdate}(V^*_i(s))$
  end
until $||V^*_{i+1} - V^*_i|| < \epsilon$;
Output: V^*_i

However, value iteration requires the tabular representation of an MDP that perfectly describes the RL problem. Extracting an accurate MDP is usually unrealistic, especially for RL problems closer to real-world tasks. The reason is that the dynamics can be too complex to be formalised by the transition and reward functions, such as when there are multiple agents or unobservable states. The Markov property of memorylessness is also a fundamental assumption that one may not satisfy. The difficulty in MDP modelling explains why, although an optimal algorithm exists, RL remains a challenge to the field.

2.1.2 Graph Neural Networks

An MDP can be modelled as *graph-structured data*, by treating states s as nodes and state transitions $s \xrightarrow{a} s'$ as edges. On the other hand, Graph Neural Networks (GNNs) [16][17] were introduced to generalise traditional deep learning techniques, such as stacking multiple layers of convolutions, onto graph-structured data, preserving the intrinsic graph information like the topological dependencies between nodes.

Formally, a graph can be formulated as $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node $s \in V$ is connected to a neighbour $s' \in \mathcal{N}(s) \subseteq V$ by an edge $e_{s' \to s} \in E$. We can then represent high-dimensional data as node features $\vec{h}_s \in \mathbb{R}^m$ and edge features $\vec{e}_{s' \to s} \in \mathbb{R}^n$, where m and n are dimensions.

Message-passing GNN:

One flavour of GNNs is the message-passing GNN [18], which iteratively updates its node feature $\vec{h}_s$ by aggregating *messages* from its neighbouring nodes as illustrated in Figure 2.1.2.

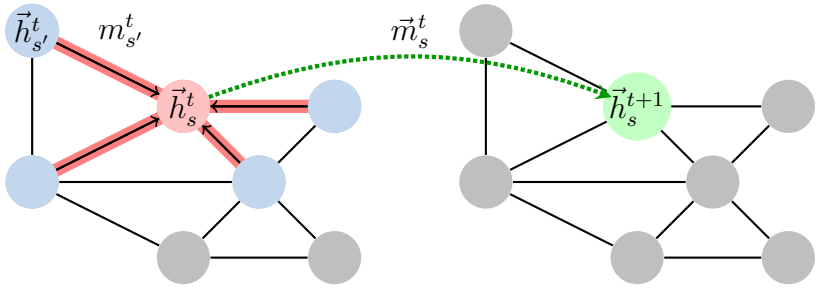


Figure 2.1.2: In one timestep of message-passing, a node gathers messages from its neighbours (left) and updates its node features using the aggregated message (right).

Formally, at each timestep t, a message can be computed between each connected pair (s, s') by a message function $M(\vec{h}^t_s, \vec{h}^t_{s'}, \vec{e}_{s' \to s})$. A node receives messages from all its connected neighbours and aggregates the messages via an operator $\bigoplus$. Since GNN operates on graphs, the ordering of neighbours may not be defined. Therefore the aggregate operator $\bigoplus$ has to be permutation-invariant, which means it produces the same output regardless of the spatial permutation of the inputs. The aggregated message $\vec{m}^t_s$ of a node s from all its neighbours $\mathcal{N}(s)$ is formulated as:

$$\vec{m}^t_s = \bigoplus_{s' \in \mathcal{N}(s)} M(\vec{h}^t_s, \vec{h}^t_{s'}, \vec{e}_{s' \to s}) \tag{2.1.6}$$

The node feature $\vec{h}^t_s$ is then transformed via an update function U:

$$\vec{h}^{t+1}_s = U(\vec{h}^t_s, \vec{m}^t_s) \tag{2.1.7}$$

2.1.3 Neural Algorithmic Reasoning

We have seen that value iteration uses MDP to compute the optimal policy and its state-value function, while GNN can operate on MDP data if we model MDP as graphs. It is natural to wonder if we can link GNN and value iteration together. Recent advances in GNN have led to Neural Algorithmic Reasoning [7], with the idea of using GNNs to learn classical Dynamic

Programming (DP) algorithms, such as value iteration. An algorithm alignment framework [8] was proposed to demonstrate the similarities between the two.

Algorithm alignment framework:

A DP algorithm decomposes the problem into smaller sub-problems, and recursively computes the optimal solutions via a DP-Update operation on sub-solutions Answer$[t][j]$.

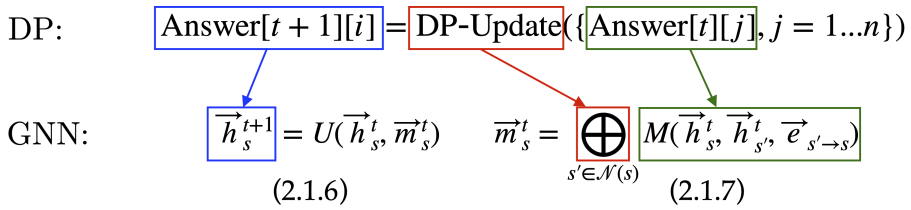


Figure 2.1.3: Alignment between DP and GNN. Here, convolution rules from a message-passing GNN are used to illustrate the similarity.

As shown in Figure 2.1.3, we can treat (i) Answer$[t][i]$ as GNN's node feature $\vec{h}_s$ and (ii) DP-Update as GNN's aggregation operator $\bigoplus$. Therefore updating Answer$[t][i]$ to Answer$[t+1][i]$ can be seen as updating GNN's node feature from $\vec{h}_s^t$ to $\vec{h}_s^{t+1}$.

Following this, they found that GNNs can simulate DP algorithms well with good sample efficiency and generalisation ability due to their close alignment. Also, [9] showed that imitating individual steps of graph algorithms using GNNs has transferability benefits.

Simulate Value Iteration with GNN:

Value Iteration (VI), being a dynamic programming algorithm, also exhibits a close alignment with GNN. Previous work [19] used a message-passing GNN to simulate each small step of value iteration by matching the Bellman optimality equation (Eqn 2.1.4) with GNN convolution steps (Eqn 2.1.6)(Eqn 2.1.7) as illustrated in Figure 2.1.4. They demonstrated that it could approximate value functions well, and display good generalisation ability on out-of-distribution data.

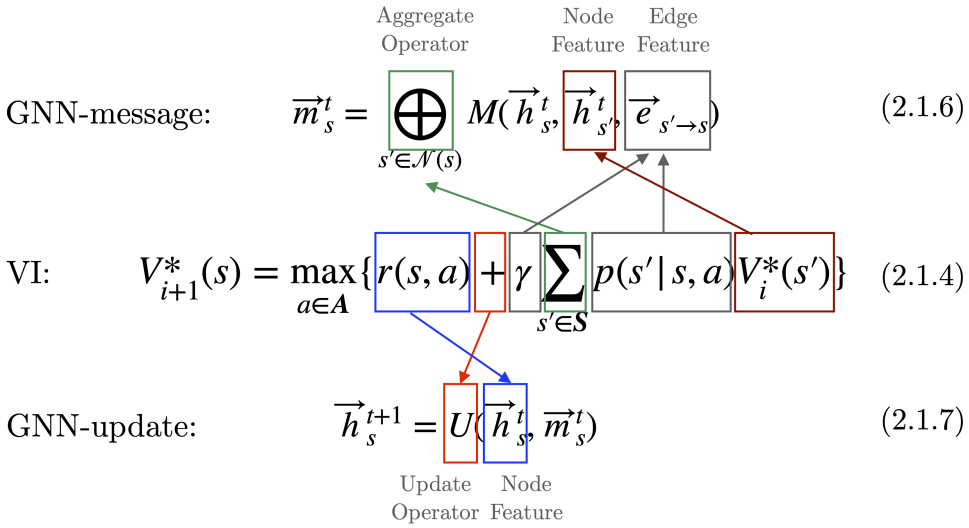


Figure 2.1.4: Alignment between Value Iteration update rule and GNN convolution rules. Pairs of correspondance are highlighted in the same colours.

Combining Neural Algorithmic Reasoning and Deep RL:

Recall that the problem of using value iteration to solve RL problems lies at the requirement of a tabular MDP that exactly matches the RL dynamics, which is usually intractable. Here, the *generalisation ability* from Neural Algorithmic Reasoning comes as a solution for filling up the gap between abstract and natural data.

Figure 2.1.5 illustrates the three components of Neural Algorithmic Reasoning framework [7]:

1. Encoder: maps a natural input into the abstract domain (left red arrow).
2. Processor: a GNN pre-trained on synthetic abstract data (blue arrows) to simulate a dynamic programming algorithm in the abstract domain.
3. Decoder: maps the abstract output back to the natural domain (right red arrow).

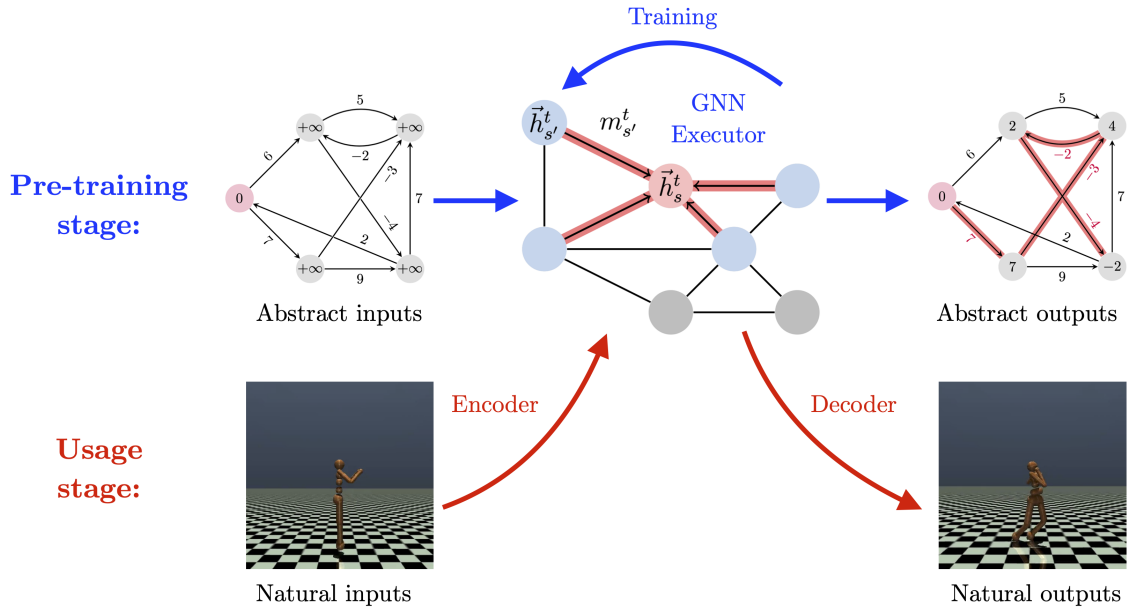


Figure 2.1.5: Framework of Neural Algorithmic Reasoning

The idea of eXecuted Latent Value Iteration Networks (XLVINs) [11] is based on Neural Algorithmic Reasoning. It trains a message-passing GNN that simulates value iteration algorithm under supervised settings with artificially prepared abstract MDP data. Then it uses the pre-trained GNN as the processor, and applies it to a "black-box" RL problem where MDP is unknown. It thus takes natural observation from the environment and outputs a policy directly.

2.1.4 Proximal Policy Optimisation

Finally, we need a training algorithm for the models. Proximal Policy Optimisation (PPO) [20] is a policy-gradient optimisation algorithm for RL problems. It is attractive for its empirically better sample complexity, simplicity, and compatibility with Stochastic Gradient Descent (SGD) [21]. An in-depth explanation of PPO can be found in Appendix A.1 due to its non-essentiality to this project's core contribution. At the high level, it performs a local search for the best policy based on policy gradients, computed with the following objective:

$$L_{\text{PPO}}(\theta) = L^{\text{CLIP+VF+}S}(\theta) = \mathbb{E}[L^{\text{CLIP}}(\theta) - c_{vf}L^{\text{VF}}(\theta) + c_s S[\pi_\theta](s)] \tag{2.1.8}$$

where $L^{\text{CLIP}}(\theta)$ estimates expected cumulative rewards, $L^{\text{VF}}(\theta)$ computes state-value function loss, and $S[\pi_\theta](s)$ measures entropy, with c_{vf} and c_s as coefficients.

The algorithm has an iterative policy evaluation-improvement style that alternates between two stages: (1) policy evaluation: perform under the current policy to generate action data, and (2) policy improvement: evaluate the loss function and then update the policy.

Algorithm 2 PPO [20]

for *i=1,2,...,rollouts* **do**
  for *actor=1,2,...,N* **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end
  Optimise $L^{\text{CLIP+VF+}S}(\theta)$, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end

2.2 Starting Point

Concepts: My relevant knowledge on the topic came from Artificial Intelligence (IB), where I learnt about multilayer perceptron and backpropagation in neural networks. Deep Neural Networks (II) and Machine Learning and Bayesian Inference (II) were also helpful, but both courses came later in the year. Prior to the project, I read the book *Reinforcement Learning: An Introduction* [1] during the summer to back up myself with the theoretical knowledge. Furthermore, XLVIN (Deac et al. 2021 [11]) is a very recent work that involves much novelty in its topics covered, and my work was to find possible solutions to extend it. Consequently, I also read current literature extensively for a wider understanding and methodology inspiration.

Tools and code: I had neither experience building neural networks nor reinforcement learning models, so I had to read up on the relevant library usage, especially PyTorch, such as its automatic differentiation with computation graphs. Besides the functionalities provided by the third-party libraries listed in Table 2.4.1, I implemented all the models and training/testing pipelines on my own, except that I adapted the PPO implementation from [22] with modifications to accommodate model, environment, and data collection pipeline differences.

2.3 Requirement analysis

The requirements stay the same as in the Project Proposal (Appendix B). The core project aims to implement and evaluate the XLVIN model in discrete control tasks, and then extend it to solve continuous control problems. For extensions, I aim to tackle XLVIN's inability in dealing with multi-dimensional and large action spaces. Complex continuous control is an active research area, therefore the extensions are exploratory in nature, requiring substantial research and experimentation on possible solutions.

In Table 2.3.1, I list out the main deliverables along with their risk analysis for better project management. Then in Figure 2.3.1, I break down the project into implementation modules and present a dependency analysis.

Main Deliverables	Priority	Risk
Environment interfaces and training/testing pipelines	High	Low
Message-passing GNN to simulate value iteration	High	Low
PPO Baseline model	High	Low
XLVIN model for discrete control problems	High	Medium
Extended model for continuous control problems	Medium	Medium
Extended model for multi-dimensional control problems (*)	Low	High
Extended model for complex continuous control problems (*)	Low	High
Open-source the implementation as a package (*)	Medium	Low

Table 2.3.1: Main deliverables and their risk analysis, where (*) indicates extensions

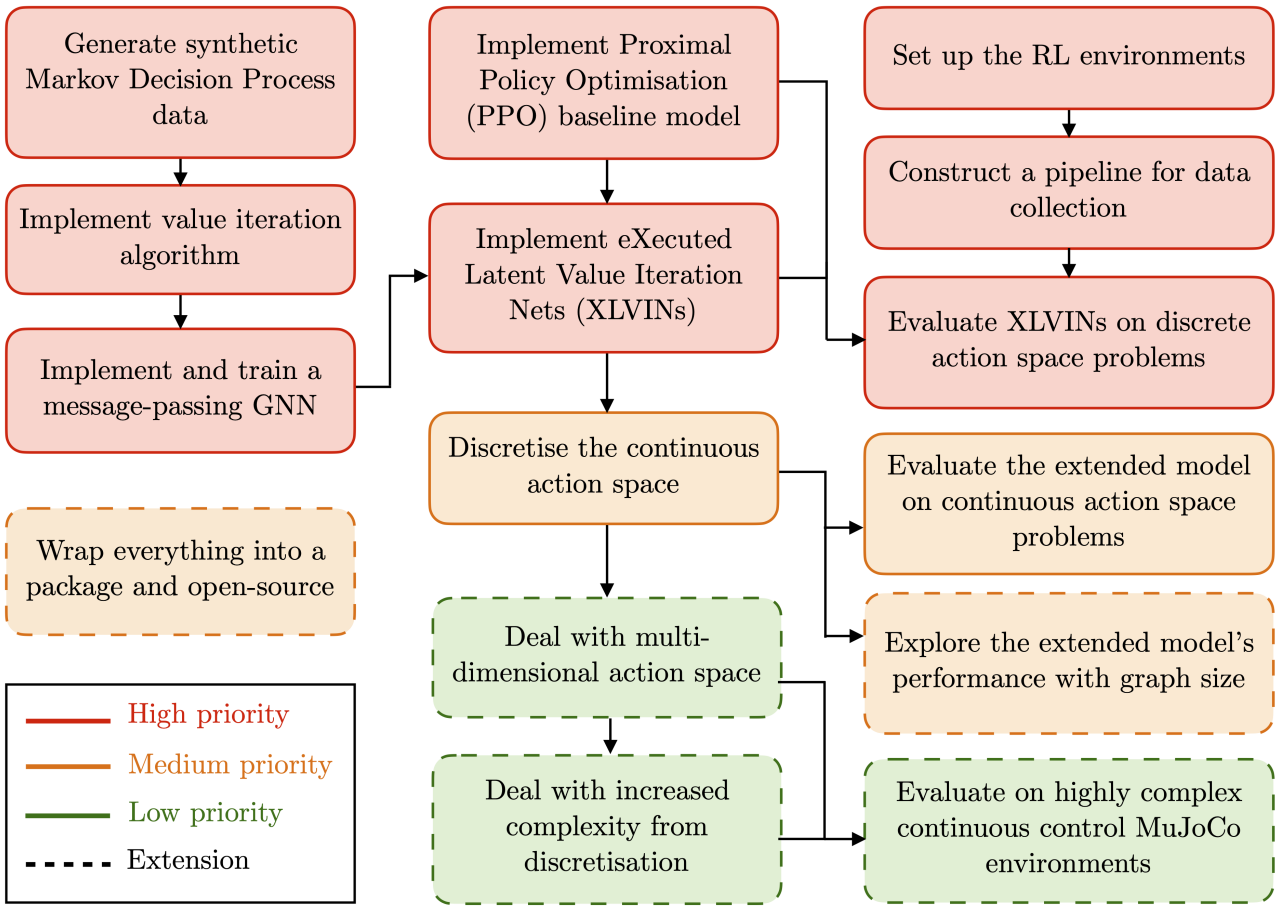


Figure 2.3.1: Dependency analysis of the implementation components

2.4 Software Engineering Techniques

2.4.1 Development model

The project is machine learning in nature, so I adopted the agile development model. Along with the modular dependency analysis in Figure 2.3.1, agile development's iterative cycle allowed me to proceed with the modules sequentially. It also provided more flexibility when the tasks became research and exploratory so that I could adapt more quickly due to the higher risks involved. Throughout the project, I also maintained good software engineering practices, such as careful modular and object-oriented design with comprehensive annotations to ensure code

readability, reusability, and reachability. I used a Gantt chart (Figure 2.4.1) for project planning with sufficient buffer time to accommodate unforeseeable difficulties, as well as Github's Kanban board for better project and code management by listing and categorising tasks in a card-style visualisation.

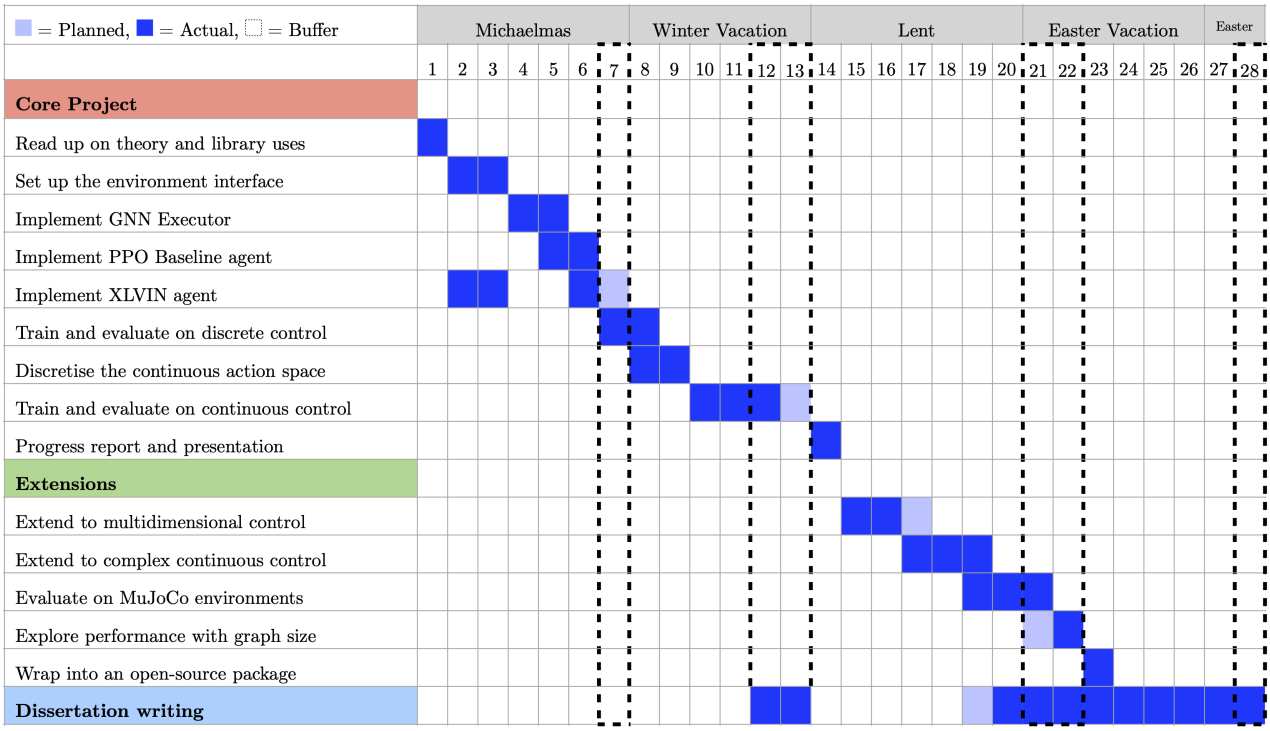


Figure 2.4.1: Gantt Chart for the project

2.4.2 Languages, libraries, tools

The language used for the project was Python 3.7 due to the employment of PyTorch, which provided extensive support for building and training neural networks. I listed the third-party libraries below and open-sourced my implementation under MIT Licence.

Library	Purpose	Licence
`torch`	Build and train neural networks; GPU interface	BSD License
`networkx`	Provide graph structures for GNN training	BSD License
`gym`	Provide Reinforcement Learning environments	MIT License
`stable_baselines3`	Provide functional wrappers around environments to facilitate training and testing	MIT License
`numpy`	Facilitate empirical studies	BSD License
`seaborn`	Provide tools for plotting and visualisation	BSD License
`matplotlib`	Provide tools for plotting and visualisation	PSF License
`setuptools`	Build into a package and open-source	MIT License

Table 2.4.1: List of third-party libraries used in the project

2.4.3 Hardware, version control, backup

I developed the code with PyCharm IDE on my personal laptop (MacBook Pro 2019, 1.4 GHz Quad-Core Intel Core i5, 8 GB). Training and testing were performed on Google Colab's GPUs (Tesla V100-SXM2-16GB). I used GitHub for version control with clear commits to easily trace back earlier versions. I also used Google Drive to backup all logs and experiment results regularly. Lastly, I used Overleaf for the writing of this dissertation in LaTeX.

Chapter 3

Implementation

This chapter explains the implementation details for the project. I start with introducing the environment interfaces (3.1), which sets up the context of problems to solve. Then I explain the idea of a GNN Executor (3.2), which is a message-passing GNN trained to simulate value iteration algorithm. The model to implement for this project is XLVIN (3.3), which I break down into four components and explain each in detail, including how it uses the GNN Executor as a core processor. This is followed by a discussion on XLVIN's limitations and how I introduced new extensions to tackle each bottleneck, leading to a new model named CNAP (3.4), which can run in complex continuous control environments. Next, I explain the training and testing strategies (3.5) for the above models. Finally, I give an overview of the code repository (3.6).

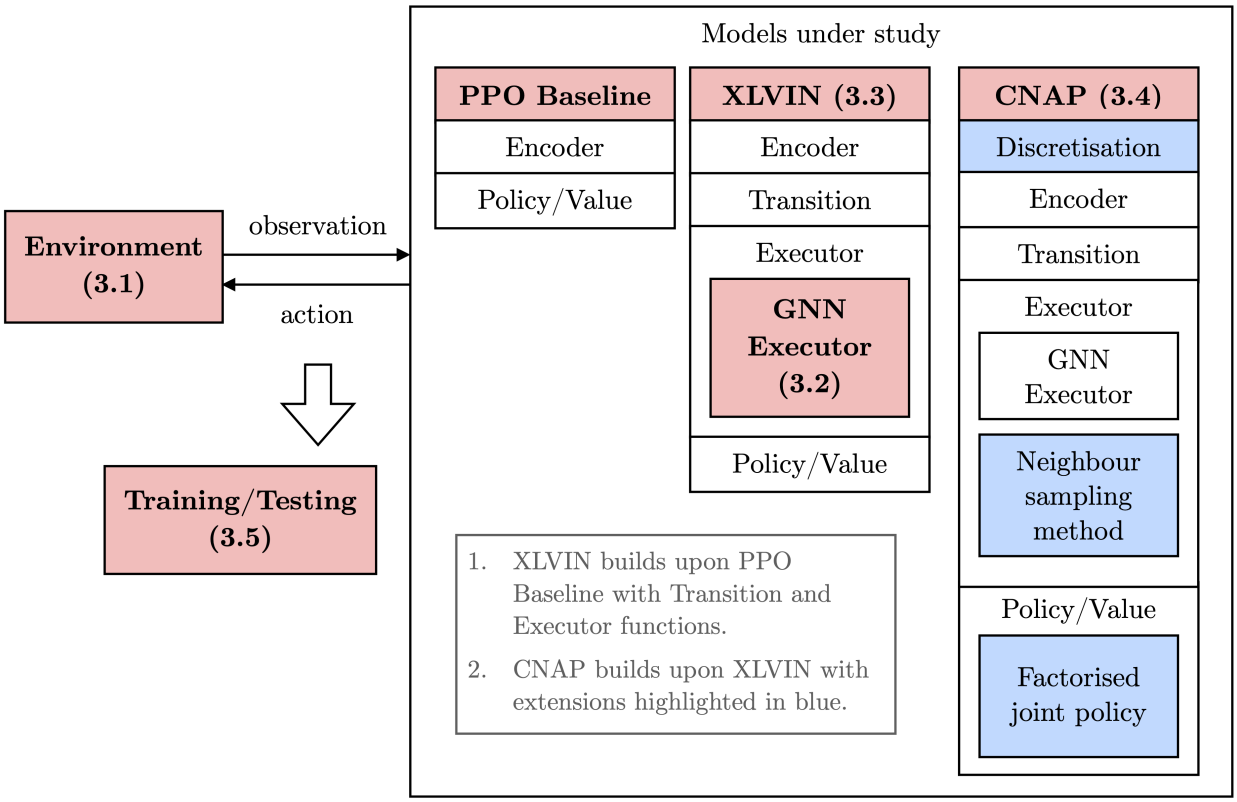


Figure 3.0.1: Overview of implementation components

3.1 Environments

An RL environment defines the problem to solve, in which an agent interacts with it to achieve a goal. I used three sets of RL environments: Classic Control suite and MuJoCo suite from OpenAI Gym [23], and a self-constructed Maze environment. Below, I explain how I set up the environment interfaces to facilitate training and testing.

3.1.1 OpenAI Gym

Take CartPole-v0 environment (Figure 3.1.1a) as an example: a pole is loosely attached to the cart by a joint. At each timestep, an agent applies a left (action=0) or right (action=1) force to the cart. It then receives a reward of +1 if the pole does not fall from the cart. The goal is to keep the pole straight as long as possible, i.e. maximising the cumulative rewards in an episode.

The key interfaces provided by OpenAI Gym are listed here with CartPole-v0 as an example:

- `env=gym.make("CartPole-v0")`: Create a new environment.
- `env.seed(seed)`: Set the environment seed for reproducibility.
- `env.reset()`: Retart a new episode of execution.
- `next_state, reward, done, infos = env.step(0)`: Pass an action=0 (pushing left) to the environment, and receive (i) `next_state` it transitions to; (ii) `reward` as a result of this action; (iii) whether the current episode is `done`; (iv) additional environment `infos`.
- `env.render()`: to visualise the current state in Figure 3.1.1a.

Additionally, I adopted functional wrappers for `env` from [24] to aid faster training and testing, as well as debugging and interpretability:

- `VectoriseWrapper`: Enabling the parallel simulation of multiple environments.
- `NormaliseWrapper`: Normalising the environment for complicated environments.
- `MonitorWrapper`: Logging additional information.
- `VideoRecorderWrapper`: Recording videos of the environment to visualise the execution.

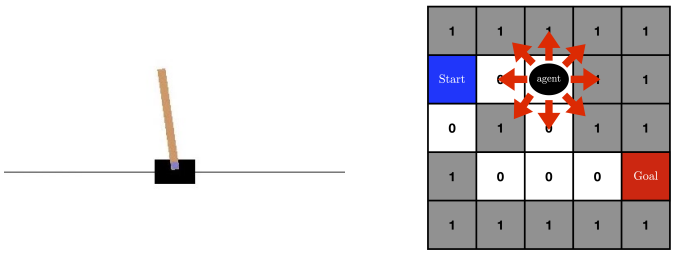


Figure 3.1.1: (a) CartPole-v0 environment (b) Maze environment

3.1.2 Self-constructed Maze

I also self-constructed a Maze environment (Figure 3.1.1b), which consists of a square grid where each cell is a road (0) or a wall (1). The agent starts at a specified position, aiming to reach a goal cell. At each timestep, it chooses to proceed in eight directions.

The Maze problem is not provided in OpenAI Gym, so I implemented a `MazeEnv` class that inherits from `gym.Env` and follows the same interfaces:

- `env=gym.make("MazeEnv")`: Call the constructor of `MazeEnv` to set up the action space (8 directions) and observation space (square grid).
- `env.reset()`: Restart the episode by loading a new maze from pre-processed grid data. Set up the start position and state.

- `next_state, reward, done, infos = env.step(0)`: Move the agent in direction 0 by one step. If the agent bumps into a wall or visits the same cell, it returns a reward of -1 and restarts the episode. If the agent reaches goal position, it returns a reward of +1 and restarts. Otherwise, it gives a reward of -0.01 to penalise prolonged episode, returns the new position, and continues execution.
- `env.render()`: A textual representation of current state.

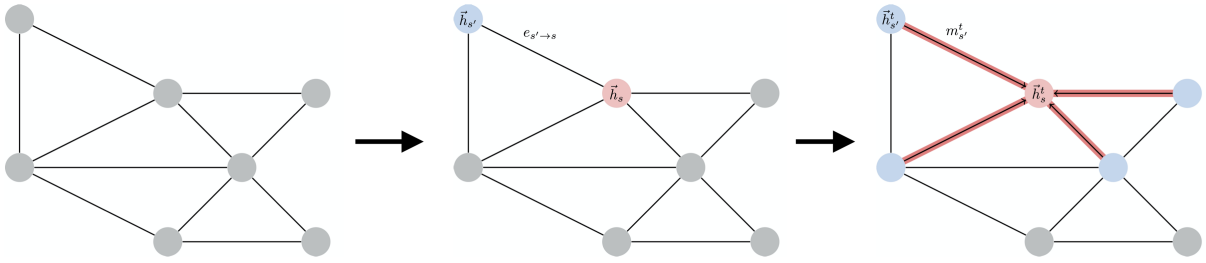
The mazes are ranked by difficulty levels: defined as the length of the shortest path from start to goal cell. I used the 8*8 grid-world data[1] consisting of 34944 mazes, computed the shortest solution path for each maze, and grouped them by difficulty levels.

3.2 GNN Executor

With the problem context set out, I now explain the implementation of the models. Recall that the central processor of XLVIN is a message-passing GNN (2.1.2) pre-trained to simulate the value iteration algorithm (2.1.1) under supervised settings. In this section, I explain how such a GNN Executor is implemented and trained.

Modelling MDP with graphs:

As discussed, value iteration takes a Markov Decision Process (MDP) (2.1.1) as input, and produces the optimal policy π^* and its state-value function V^* in an iterative update manner. Since GNN operates on graph-structured data, we need to model MDP with graphs. One choice is to have a separate graph for each action $a \in \mathcal{A}$, so we would have $|\mathcal{A}|$ graphs in total. For each graph $G_a(V, E)$, where V is the set of nodes and E is the set of edges, we represent each node as a state $s \in V$, and each edge $e_{s,s'} \in E$ as a state transition $s \xrightarrow{a} s'$.



(1) Generate graph structure (2) Populate with MDP data (3) Train a message-passing GNN

Figure 3.2.1: Training a GNN Executor

Generate graph structure:

As shown in Figure 3.2.1, the first step is to generate graph structures. I used a set of synthetic graphs with different types following previous work [25][26] (full list in Evaluation). Take Erdős-Rényi graphs as an example: given a graph size N and a node degree D, we connect any pair of nodes randomly by a probability $p = N/D$. Here, we define the graph size to be the number of states $N = |\mathcal{S}|$ and node degree to be the number of actions $D = |\mathcal{A}|$. By randomly connecting edges, the generated graph thus defines all transitions $s \xrightarrow{a} s'$.

[1]Data is taken from `https://github.com/kentsommer/pytorch-value-iteration-networks`

Populate with MDP data:

The second step in Figure 3.2.1 populates the graph with MDP data. An MDP consists of a transition probability $p(s'|s,a)$ and a reward function $r(s,a)$. Following the alignment between Value Iteration update rule and GNN convolution rules in Figure 2.1.4, we represent MDP data using node features and edge features:

$$\text{node feature:} \quad \vec{h}(s) = (V(s), r(s,a)) \tag{3.2.1}$$

$$\text{edge feature:} \quad \vec{e}(s,s') = (\gamma, p(s'|s,a)) \tag{3.2.2}$$

where $\gamma \in [0,1]$ is the discount factor that determins how much emphasis to put in the distant future rather than the immediate future. We also include the state-value function $V(s)$ in the node feature as our target predictions.

We can now generate MDP data to fill up the node features and edge features. γ is the discount factor, which is a fixed hyperparameter value. $V(s)$ is the value function, so I implemented the value iteration algorithm to calculate the ground-truth values. Finally, I randomly generated the values for $p(s'|s,a) \in [0,1]$ and $r(s,a) \in [-1,1]$, with probability constraint $\sum_{s' \in S} p(s'|s,a) = 1$.

Train a message-passing GNN:

The final step in Figure 3.2.1 is the training. As described in Section 2.1.2, at each timestep t, a node in a message-passing GNN (i) gathers messages from all its neighbouring nodes, (ii) aggregates messages using a permutation-invariant operator, and (iii) updates its own node feature.

(i) We define the message function M to be a fully-connected linear layer. For each pair of neighbours (s, s'), we use M to compute a message using their node features and edge feature:

$$m(s,s') = M(\vec{h}(s), \vec{h}(s), \vec{e}(s,s')) \tag{3.2.3}$$

(ii) Then for each node, we aggregate the messages from all its neighbours by summation, which is permutation-invariant:

$$\vec{m}(s) = \sum_{s' \in \mathcal{N}(s)} m(s,s') \tag{3.2.4}$$

(iii) Lastly, we update the node's own feature vector by adding the aggregated message:

$$\vec{h}(s) = \vec{h}(s) + \vec{m}(s) \tag{3.2.5}$$

Previous work [9] has found generalisation benefits when using GNNs to simulate individual steps of algorithms. Therefore, we train the GNN Executor to mimic the residual from one step of value iteration update, i.e. $\delta = V_{t+1}(s) - V_t(s)$. Value iteration iteratively optimises $V(s)$, so the GNN is instructed to perform the same number of update steps as the ground-truth value iteration does.

The message-passing GNN is trained under supervised settings, where the loss function is the Mean Squared Error (MSE) of the predicted value function:

$$L_{\text{MSE}} = \sum_{s \in \mathcal{S}} (V_{true}(s) - V_{pred}(s))^2 \tag{3.2.6}$$

where V_{true} is computed using value iteration, and V_{pred} is from the GNN output.

3.3 eXecuted Latent Value Iteration Nets (XLVINs)

The GNN Executor is the central processor of XLVIN, which is the model that I aim to reproduce. XLVIN adopts the Neural Algorithmic Reasoning framework: encode, process, decode. The GNN Executor is the processor, approximating the optimal policy's state-value function. However, XLVIN also needs to perceive observation from the environment (encode) and instruct the agent to perform actions (decode). In this section, I decompose XLVIN into four components and explain their designs individually.

Here is an overview of the four components:

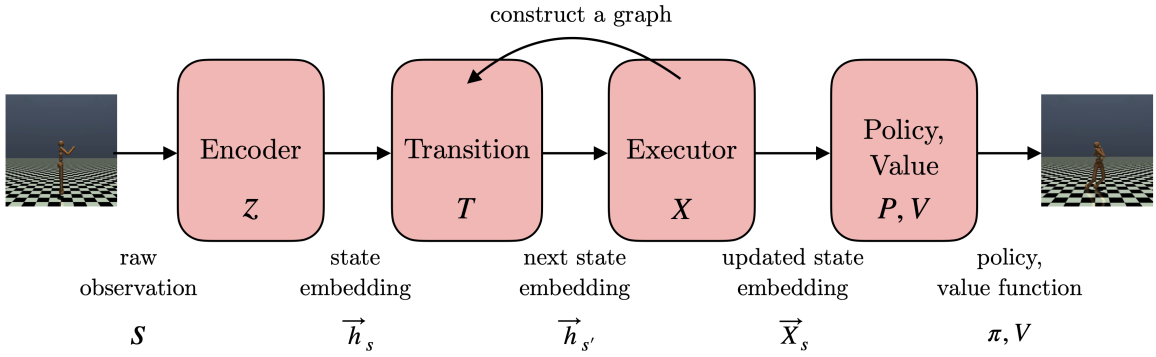


Figure 3.3.1: Framework of XLVIN

3.3.1 Encoder and Transition

Encoder:

`Encoder` ($z : \mathcal{S} \to \mathbb{R}^k$) maps a raw observation from the environment $s \in \mathcal{S}$, to a state embedding in a latent space $\vec{h_s} = z(s) \in \mathbb{R}^k$. The dimension of a state embedding is a hyperparameter k. The latent space is a low-dimensional vector space where the distance between state embeddings reflects similarity, that is, more similar ones are grouped closer together.

`Encoder` is implemented using a three-layer Multilayer Perceptron (MLP) with an input layer, a hidden layer, and an output layer. An MLP is a fully-connected feedforward network, as shown in Figure 3.3.2, where there is a single direction of signal flow from input to output. The Rectified Linear Units (ReLU) [27] activation function ($\phi(x) = \max(0, x)$) is inserted between each layer so that the network can distinguish data that is not linearly separable.

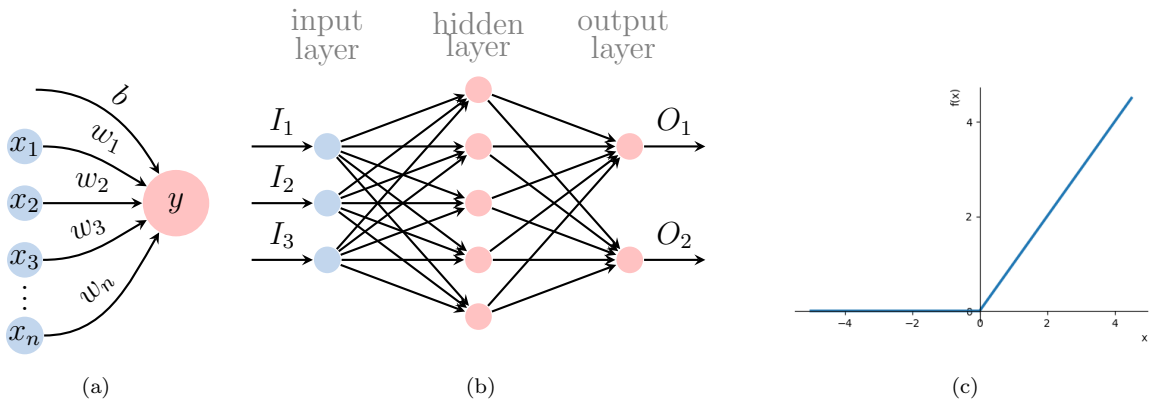


Figure 3.3.2: (a) a single perceptron (b) a 3-layer MLP (c) non-linearity of ReLU activation

Transition:

`Transition` ($T : \mathbb{R}^k \times \mathcal{A} \to \mathbb{R}^k$) takes two inputs: the state embedding of an observation $z(s) \in \mathbb{R}^k$, and an action $a \in \mathcal{A}$. It predicts the transition leading to the next state embedding $z(s') \in \mathbb{R}^k$, where ideally the environment would experience a transition $s \xrightarrow{a} s'$. Formally,

$$z(s) + T(z(s), a) \approx \mathbb{E}_{s' \sim P(s'|s,a)} z(s') \tag{3.3.1}$$

The action a is flattened using one-hot encoding and stacked together with the state embedding $z(s)$ as one input vector. `Transition` has the same architecture as `Encoder`, using a three-layer MLP with ReLU added between each layers. The difference is that Layer Normalisation [28] is applied before the output layer to normalise the input distribution for smoother gradients and better generalisation ability:

$$y = \frac{\boldsymbol{x} - \mathbb{E}[\boldsymbol{x}]}{\sqrt{\mathbb{V}\text{ar}[\boldsymbol{x}] + \epsilon}} + \beta \tag{3.3.2}$$

Training with contrastive TranseE loss:

The performance of `Encoder` and `Transition` relies on the mapping from raw observations to state embeddings in the latent space. Ideally, if s' is the successor state when taking an action a under state s, then we want to achieve two goals: (i) group $\vec{h}_{s'}$ as close as possible to $\vec{h}_s$, while (ii) keeping $\vec{h}_{s'}$ away from any other state embedding $\vec{h}_{\tilde{s}}$. To achieve (i), the loss function is designed based on TransE [29], which was proposed to model relationships as translations for entity embedding tasks. To achieve (ii), it is augmented with contrastive learning [30], where negative samples $\tilde{s}$ are also taken into account. A negative sample is a state $\tilde{s}$ that is different from the true next state s'. I stored all transition triplets (s, a, s') experienced by the agent in a replay buffer and randomly sampled the negative sample $\tilde{s}$ from it using a uniform distribution.

Formally, the contrastive TransE loss function is defined as:

$$L_{\text{TransE}}((s, a, s'), \tilde{s}) = d(z(s) + T(z(s), a), z(s')) + \max(0, \xi - d(z(\tilde{s}), z(s'))) \tag{3.3.3}$$

where $d(x_1, x_2) = ||x_1 - x_2||_2^2$, and $\xi > 0$ is a hinge hyperparameter that defines a lower bound on the distance between two state embeddings that we would consider as different.

3.3.2 Executor

Input features:

`Executor` ($X : \mathbb{R}^k \to \mathbb{R}^k$) uses the GNN that simulates value iteration from Section 3.2 as the core processor. Recall when training the GNN, we used the feature definitions below to generate artificial MDP data with synthetic graphs:

$$\begin{aligned} \text{node feature:} \quad & \vec{h}(s) = (V(s), r(s, a)) \\ \text{edge feature:} \quad & \vec{e}(s, s') = (\gamma, p(s'|s, a)) \end{aligned}$$

However, it is usually intractable to map RL problems exactly into MDPs because the dynamics can be too complex. This means transition probability $p(s'|s, a)$ and reward function $r(s, a)$ may not be provided. We therefore use slightly different feature definitions to accommodate the lack of MDP data:

$$\text{node feature:} \quad \vec{h}(s) = z(s) \tag{3.3.4}$$

$$\text{edge feature:} \quad \vec{e}(s, s') = (\gamma, a) \tag{3.3.5}$$

where $z(s)$ is the state embedding from `Encoder` output, and a is an action.

The difference in input distribution is handled by the generalisation power from Neural Algorithmic Reasoning [8][9][19]. It hypothesises that when plugged into XLVIN, the pre-trained GNN that imitates value iteration can still predict well even though the feature definitions are different. This hypothesis will be validated in Evaluation.

Construct a GNN graph:

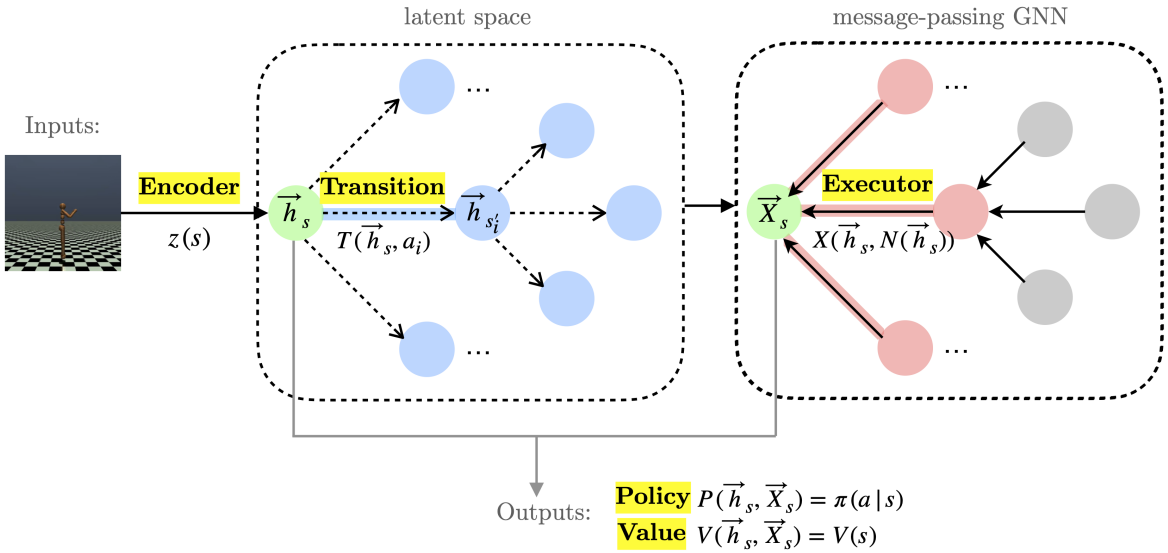


Figure 3.3.3: Given a state embedding $\vec{h}_s$ as a starting node, `Executor` uses `Transition` $T(\vec{h}_s, a_i)$ to expand edges. It builds a graph and then simulates value iteration with the pre-trained GNN.

From feature definitions (Eqn 3.3.4; Eqn 3.3.5), we can see each *node* corresponds to a *state embedding*, while each *edge* corresponds to an *action*. Therefore, we can construct a GNN graph as shown in Figure 3.3.3: (1) treat the input state embedding $\vec{h}_s$ as a starting node; (2) enumerate all possible actions $a_i \in \mathcal{A}$ as edges to expand from the current node; (3) use `Transition` to predict the next state embedding $\vec{h}_{s'_i}$ as neighbours, corresponding to the transition $s \xrightarrow{a_i} s'_i$:

$$\mathcal{N}(\vec{h}_s) = \{\vec{h}_{s'_i} = T(\vec{h}_s, a_i), \text{ for all } a_i \in \mathcal{A}\} \tag{3.3.6}$$

This adds one layer of new nodes to the GNN graph. We can repeat the process for t times, each time expanding the nodes from the previous layer, leading to a graph of t layers, and each node has a degree of $|\mathcal{A}|$. We can say the GNN graph has depth t and width $|\mathcal{A}|$. The depth t also indicates the number of value iteration updates the GNN would perform. Finally, `Executor` outputs an updated state embedding, $\vec{\mathcal{X}}_s = X(\vec{h}_s, \mathcal{N}(\vec{h}_s))$. The pseudocode for `Executor` can be found in Appendix A.2.

Architecture:

The pre-trained GNN is directly plugged in, augmented with an input layer (to process inputs received from `Encoder`) and a output layer (to process outputs before sending to `Policy/Value`), which are both implemented as single fully-connected linear layers.

3.3.3 Policy and Value

Policy:

As shown in Figure 3.3.3, `Policy` ($P : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^{|\mathcal{A}|}$) takes the outputs from `Encoder` and `Executor`, i.e. the state embedding $\vec{h_s}$ and the updated state embedding $\vec{\mathcal{X}}_s$, then produces a policy $\pi = p(a|s)$ defining the probability of taking each action a under current state s.

`Policy` is a fully-connected linear layer with an output dimension same as the number of all possible actions $|\mathcal{A}|$. The output policy $\pi \in \mathbb{R}^{|\mathcal{A}|}$ is used as logits for a categorical distribution where the categories are the actions. The agent can choose an action by sampling deterministically (i.e. taking the mode) or non-deterministically (i.e. taking a random sample) from the categorical distribution by providing the current state.

Value:

`Value` ($V : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$) takes the same inputs as `Policy`, and produces the policy's estimated state-value function of the current state $V(s)$.

`Value` is also a fully-connected linear layer with a scalar output, which is an approximated value for $V(s)$.

3.4 Continuous Neural Algorithmic Planners (CNAPs)

The main limitations of XLVINs are their incapabilities to deal with continuous control environments and large action spaces. In this section, I propose Continuous Neural Algorithmic Planners (CNAPs) and explain the extensions I made based on XLVINs to tackle the aforementioned limitations. I will demonstrate a successful application of CNAPs in complex continuous problems in Evaluation. CNAPs move the idea of XLVINs forward to a much broader application range closer to real-world domains.

3.4.1 Dealing with continuous action space

3.4.1.1 Discretisation by binning

The discretisation technique I chose was to split the continuous action space into evenly spaced discrete action bins. Previous works such as [31][32][33] have demonstrated the effectiveness of such technique with on/off-policy optimised agents in continuous RL problems. Without losing generality but for demonstration purpose, we can assume the action space to be $\mathcal{A} = [0, 1]$. The number of action bins is set as a hyperparameter N, so the action space is discretised into:

$$\mathcal{A}_{discrete} = \{a_0, a_1, ..., a_{N-1}\}, \tag{3.4.1}$$

$$\text{where } a_i = \frac{2i+1}{2N}, \text{ i.e. the median of } \left[\frac{i}{N}, \frac{i+1}{N}\right] \tag{3.4.2}$$

Instead of a continuous range, we now have N discrete actions in our action space. The above equation can be easily generalised to an arbitrary continuous range.

In OpenAI Gym, a discrete space is implemented by class `Discrete`, and a continuous space uses class `Box`, both inherited from the superclass `Space`. In order to integrate the discretisation process into XLVIN, we can implement a wrapper on a given environment to turn its action space type from `Box` to `Discrete`:

```
def discretising_wrapper(env, N):
    # discretise env's action space into N discrete actions

    def discretised_reset():
        # calling the original env's reset() function

    def discretised_step(action):
        # turn the action type from Discrete to Box
        # calling the original env's step(action) function

    # replace the original env's reset and step functions
    env.reset = discretised_reset
    env.step = discretised_step

    return env
```

With this wrapper, the agent can send discrete action to `env.step(action)` and interact with the environment as before.

3.4.2 Dealing with higher complexity

The action space experiences a combinatorial explosion in size when it expands from one-dimensional to multi-dimensional. This problem becomes almost unavoidable when a continuous action space is discretised with the above method. For example, assuming the action space $\mathcal{A}$ has D dimensions, and in each dimension, we discretise into N action bins, this leads to $|\mathcal{A}_{discrete}| = N^D$ actions in total. Real-world RL problems usually have complex dynamics with high-dimensional action space. The explosion in action space thus poses a challenge for the vanilla XLVIN models.

Specifically, there are two bottlenecks in the model design that inhibit XLVINs from acting in a large action space:

(i) `Policy` outputs an optimal policy π^* with a dimension of $|\mathcal{A}|$, because $\pi^* = p(a|s)$ specifies the probability of choosing each action given the current state.

(ii) `Executor` requires a graph where nodes are states and edges are actions. It constructs the graph by expanding a node with an enumeration of all possible actions, so each node has a degree of $|\mathcal{A}|$. The graph size also adds another level of exponential explosion as we expand the graph layer by layer from a central node.

Below, I will address the two bottlenecks respectively. I propose to use a factorised joint policy for bottleneck (i), and a neighbour sampling method for bottleneck (ii).

3.4.2.1 Factorised joint policy (Extension)

Say each action $\vec{a} \in \mathcal{A}$ has D dimensions, and each dimension has N discrete action bins. A naive policy $\pi^* = p(\vec{a}|s)$ produces a categorical distribution with N^D possible actions. To tackle this explosion in policy dimension, previous work such as [31] uses a sequence model to predict action in each dimension sequentially, but this poses a prior assumption on the order of dimensions. A parallel work [32] proposes to learn a separate policy for each dimension independently, but their results were limited in a small number of tasks. I chose to adopt the idea from [33] to use a factorised joint policy due to its promising performance on a comprehensive

set of continuous environments:

$$\pi^*(\vec{a}|s) = \prod_{i=1}^{D} \pi_i^*(a_i|s) \tag{3.4.3}$$

It approximates D policies at the same time, where each policy $\pi_i^*(a_i|s)$ indicates the probability of choosing an action $a_i \in \mathcal{A}_i$ in the i^{th} dimension, where $|\mathcal{A}_i| = N$. In total, the factorised joint policy π^* has an output dimension of $N \cdot D$ instead of N^D. Hence the dimension of π^* grows linearly with the increase of dimension, instead of exponentially.

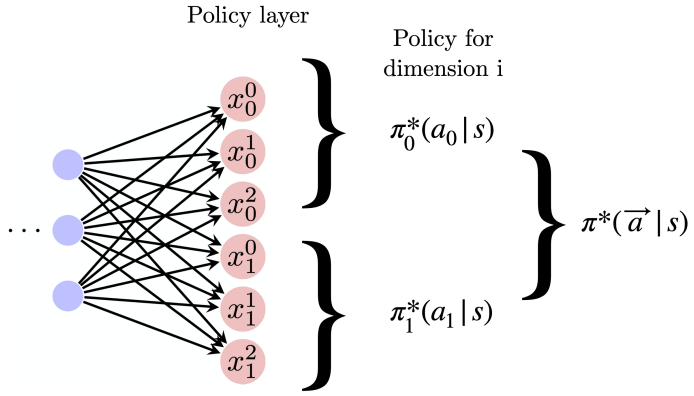


Figure 3.4.1: Factorised joint policy for N=3 and D=2

To implement this change, `Policy` layer outputs $|\mathcal{A}| = N \cdot D$ logits, and we separate these logits sequentially into D groups: $[(x_0^0, ..., x_0^{N-1}), ..., (x_{D-1}^0, ..., x_{D-1}^{N-1})]$ as shown in Figure 3.4.1. Then we send each group of logits $\vec{x}_i$ to a categorical distribution i. To sample an action $\vec{a}$ from π^*, we pick each a_i from categorical distribution i, then stack them together as $\vec{a} = [a_0, ..., a_{D-1}]^T$.

Other implementation change across the whole model is needed. This includes adapting from `Discrete` to `MultiDiscrete`, which is the class used by OpenAI Gym for multi-dimensional space, as well as the one-hot encoding method used for actions.

3.4.2.2 Neighbour sampling method (Extension)

As shown in Figure 3.4.2, the second bottleneck lies in the `Executor`. When `Executor` constructs a graph to execute the pre-trained GNN, it treats each state as a node, then enumerates all possible actions $\vec{a}_i \in |\mathcal{A}|$ to connect neighbours via approximating $\vec{h}(s) \xrightarrow{\vec{a}_i} \vec{h}(s_i')$. Therefore each node has degree $|\mathcal{A}|$, and graph size grows even faster as it expands deeper.

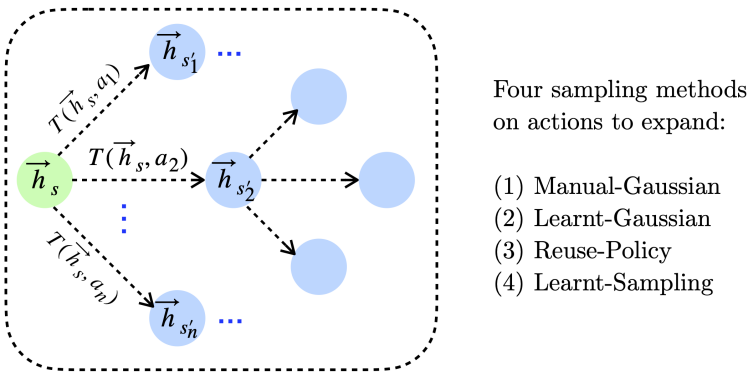


Figure 3.4.2: `Executor` constructs a graph by enumerating all possible actions, thus $n = |\mathcal{A}|$, which becomes intractable. Four sampling methods are proposed to avoid a full enumeration, so only $n = K \ll |\mathcal{A}|$ actions are selected to expand.

To tackle this challenge, instead of using all possible actions, I propose to use a neighbour sampling method to choose a subset of actions to expand. The important question is which actions to select. The pre-trained GNN uses the graph constructed to simulate value iteration behaviour and predict the state-value function. Recall the value iteration update rule:

$$V^*_{i+1}(s) = \max_{a \in \mathcal{A}} \{r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*_i(s')\}$$

Hence, it is critical that we can include the action that produces a good approximation of the state-value function in our sampling.

Four possible methods:

We set the number of neighbours to expand for each node, $K \ll |\mathcal{A}|$, as a hyperparameter. I now explain four possible methods that I propose to sample K actions from $\mathcal{A}$.

Firstly, I implemented two Gaussian-based methods for sampling. Gaussian distribution is a common baseline policy distribution for continuous control problems (such as in [33]), and it is straightforward to interpret. Gaussian distribution discourages extreme actions while encourages neutral ones with some continuity, so it can potentially be a good candidate. A Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ can be formalised as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{3.4.4}$$

where μ is the mean, and σ is the standard deviation.

(1) Manual-Gaussian: A Gaussian distribution is used to randomly sample action values in each dimension $a_i \in \mathcal{A}_i$, which are stacked together as a final action vector $\vec{a} = [a_0, ..., a_{D-1}]^T \in A$. We repeat for K times to sample a subset of K action vectors. We set the mean $\mu = N/2$ and standard devisation $\sigma = N/4$, where N is the number of discrete action bins. These two parameters are chosen to spread a reasonable distribution over $[0, N-1]$. Outliers and non-integers are rounded to the nearest whole number within the range $[0, N-1]$.

(2) Learnt-Gaussian: The two parameters manually chosen in the previous method pose a constraint on placing the median action in each dimension as the most likely. Here instead, two fully-connected linear layers are used to estimate the mean μ and standard deviation σ separately. They take the state embedding $\vec{h}_s$ from `Encoder` and output a scalar value each. The rest of sampling remains the same.

Gaussian methods still restrain a fixed distribution on the sampling distribution, which it may not necessarily fit. Previous work [34] studied a similar action sampling problem. They reasoned that since the actions selected by the policy are expected to be more valuable, we can directly use the policy for sampling. This inspired me to propose the following method.

(3) Reuse-Policy: We can reuse `Policy` layer $P(\vec{h}_s, \vec{\mathcal{X}}_s)$ to sample the actions when we expand the graph in `Executor`. This is equivalent to using the policy distribution $\pi^* = p(\vec{a}|s)$ as the neighbour sampling distribution. However, the second input $\vec{\mathcal{X}}_s$ for `Policy` layer comes from `Executor`, which is not available at the time of constructing the graph. I chose to fill up the space by setting $\vec{\mathcal{X}}_s = \vec{0}$ as placeholders.

Lastly, I tried to use a separate layer dedicated to learning the neighbour sampling distribution.

(4) Learnt-Sampling: I implemented this method with a fully-connected linear layer that consumes $\vec{h}_s$ and has an output dimension of $|N \cdot D|$. It is expected to learn the optimal neighbour sampling distribution in a factorised joint manner same as Figure 3.4.1. The outputs are logits for D categorical distributions, one for each dimension, together producing $\vec{a} = [a_0, ..., a_{D-1}]^T$.

I will provide an analysis of the above methods in Evaluation. Note that a neighbour sampling method not only targets increased complexity from discretisation in higher dimensions, but it can also solve any problem with large action spaces. Together with a factorised joint policy, these extensions tackle the aforementioned limitations of XLVINs, bridging its gap practically to continuous and higher complexity domains. I named this extended model Continuous Neural Algorithmic Planners (CNAPs) and will use this abbreviation in later chapters.

3.5 Training and Testing

With the model designs explained, I now describe the training and testing strategies.

3.5.1 Training Strategy

Loss function:

Both XLVIN and CNAP models are trained using Proximal Policy Optimisation (PPO) algorithm. The GNN Executor is pre-trained and directly plugged in, while the rest are all learnable parameters. The combined loss of the PPO objective (Eqn 2.1.8) and the contrastive TransE loss (Eqn 3.3.3) is:

$$L(\theta) = L_{\text{PPO}}(\theta) + \lambda \sum_{((s,a,s'),\tilde{s}) \in \mathcal{T}} L_{\text{TransE}}((s,a,s'),\tilde{s}) \tag{3.5.1}$$

where θ are learnable parameters, λ is the TransE loss coefficient, and $\mathcal{T}$ is the set of transitions and negative samples passed to `Transition` function.

PPO uses a policy evaluation-improvement cycle to iteratively approximate the best policy $\pi^* = p(a|s)$. The algorithm was explained with pseudocode in Section 2.1.4. I now explain how PPO training is used particularly in this project.

Policy Evaluation:

In policy evaluation phase, PPO collects a set of training data by interacting with the environment under the current policy $\pi(\theta)$. A set of training environments are vectorised to allow parallel execution. We initiate state transitions by `env.step(action)`, where `action` is chosen by the current policy $\pi(\theta)$. We then store data collected (`state`, `next_state`, `reward`, `done`) into a buffer. Training is repeated until a fixed number of episodes or steps are reached.

Policy Improvement:

In policy improvement phase, the loss function is computed to update the model parameter θ to θ' with Stochastic Gradient Descent (SGD) [21]. SGD is used here due to its data efficiency. It randomly splits training data from the buffer into minibatches, and computes the loss function (Eqn 3.5.1) for each minibatch. It then takes a small step in the negative direction of the gradient with respect to the model parameter in order to minimise the loss function:

$$\theta' = \theta - \eta \frac{\partial L_{mini}(\theta)}{\partial \theta}, \text{ with step size } \eta \tag{3.5.2}$$

Now the old policy $\pi(\theta)$ is replaced with $\pi(\theta')$. After SGD repeats for a few epochs, the policy evaluation-improvement cycle can repeat for a fixed number of iterations.

Optimiser:

The optimiser choice is Adam optimiser [35], a commonly used variant of the vanilla SGD algorithm. It improves over plain SGD by incorporating (i) a momentum term that accumulates and averages recent gradients to counter noisy gradient updates and (ii) an adaptive learning rate to cater for different gradient movement speeds across dimensions. Additionally, I also augmented with an optional learning rate decay that decreases the learning rate linearly with PPO iteration. A learning rate decay can stabilise gradients and avoid oscillatory behaviour, especially when a much longer training process is needed for complex environments.

3.5.2 Testing Strategy

The testing procedure is similar to the policy evaluation phase. A different set of testing environments are created, and the agent behaves under the approximated policy $\pi(\theta)$. Rewards are collected for a fixed number of episodes and averaged per episode. The goal of the model is to maximise this mean episodic reward.

3.6 Repository Overview

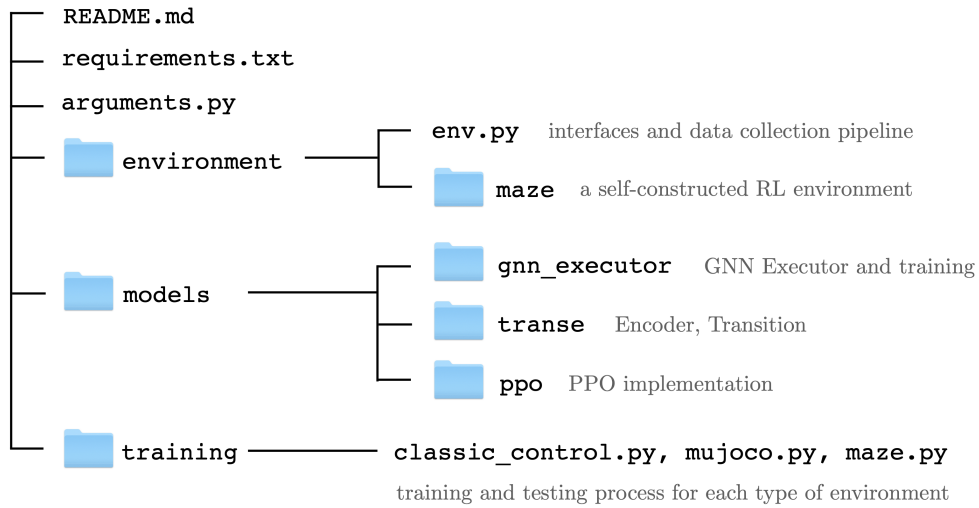


Figure 3.6.1: Structure of the repository

Figure 3.6.1 presents the repository structure. Due to the difficulty in unit testing Deep RL, I mainly used the testing strategy described above in Section 3.5.2 and checked if the results align with existing literature, such as [11] for Classical Control and [33] for MuJoCo, to validate the model implementation.

Chapter 4

Evaluation

In this chapter, I present the quantitative and qualitative evaluations performed to demonstrate how the success criteria of core project and extensions are met. The main topics to explore are:

- *How well can GNN Executor simulate value iteration algorithm? (4.1)*
- *How well can XLVINs perform in discrete control environments? (4.2)*
- *How well can CNAPs perform in continuous control environments? (4.3)*
- *How well can CNAPs perform in highly complex continuous control environments? (4.4)*

Model	Evaluation Axes	Environments
GNN Executor	Accuracy, Generalisation Ability	N.A.
XLVIN	Performance, Stability, Data Efficiency	Discrete Classic Control
	Predictability	Self-constructed Maze
	Generalisation Ability	
CNAP	Performance, Stability, Data Efficiency	Continuous Classic Control
	Effect of graph width	
	Effect of graph depth	
CNAP	Scalability	Continuous MuJoCo
	Interpretation Analysis	

Table 4.0.1: Overview of the evaluation axes under inspection

Both XLVIN and CNAP were compared with a PPO Baseline agent. PPO is a state-of-the-art agent used as default at OpenAI due to its competitive performance. For reproducibility purpose, the complete list of hyperparameter values is attached in Appendix A.5.

4.1 GNN Executor

The GNN Executor is a message-passing GNN trained to imitate each small step of value iteration algorithm. To guide the evaluation, we can ask the following questions:

Qn. How well can the GNN Executor simulate value iteration algorithm?

- *Can it predict the optimal policy and state-value function accurately? (Table 4.1.1)*
- *Can it generalise well to out-of-distribution data? (Table 4.1.2 and 4.1.3)*
- *Can it scale up efficiently? (Table 4.1.3)*

Experiment setup:

The GNN Executor outputs an approximated optimal policy $\pi(a|s)$ and its state-value function $V(s)$. The goal is to estimate the two outputs as close as possible to value iteration's ground-truth outputs. Hence, below I define two quantitative metrics to evaluate the predictions:

- Mean squared error (MSE) of state-value function $V(s)$:

$$\text{MSE} = \sum_{s \in \mathcal{S}} (V_{true}(s) - V_{pred}(s))^2 \tag{4.1.1}$$

- Accuracy of policy $\pi(a|s)$:

$$\text{Accuracy} = \left(\sum_{s \in \mathcal{S}} f(\underset{a_{true} \in \mathcal{A}}{\text{argmax}}\, \pi_{true}(a_{true}|s), \underset{a_{pred} \in \mathcal{A}}{\text{argmax}}\, \pi_{pred}(a_{pred}|s)) \right) / |\mathcal{S}| \times 100\%, \tag{4.1.2}$$

$$\text{where } f(a_{true}, a_{pred}) = \begin{cases} 0, & a_{true} \neq a_{pred} \\ 1, & a_{true} = a_{pred} \end{cases} \tag{4.1.3}$$

The training data was generated using 500 random synthetic Erdős-Rényi graphs with size ($|\mathcal{S}| = 20, |\mathcal{A}| = 5$). We define the number of states $|\mathcal{S}|$ to be the number of nodes, and the number of actions $|\mathcal{A}|$ to be the node degree. Each graph is populated with MDP data as described in Section 3.2.

The testing data was generated on a set of different graph types based on previous works [25][26], covering a range of sparse and dense graphs. The full description for each graph type can be found in Appendix A.3. The test graphs were also generated with different sizes by varying the numbers of states and actions, ($|\mathcal{S}|, |\mathcal{A}|$). The aim was to test the generalisation ability of the GNN on out-of-distribution data. Each test set contains 40 randomly generated test graphs.

Discussion of results:

(i) Accuracy: on similar graph types

Graph Type	$\|\mathcal{S}\| = 20,\ \|\mathcal{A}\| = 5$	
	MSE	Accuracy
Erdős-Rényi	0.57 ± 0.7	98.1 ± 2.7
Barabási-Albert	0.47 ± 0.5	98.6 ± 2.2

Table 4.1.1: MSE of state-value function and Accuracy of policy by running one step of message-passing in GNN. These graphs share similar graph structures and the same size as the train graphs. The low MSE and high Accuracy show a close alignment between the predictions and the ground-truth values.

(ii) Generalisation ability: on different graph types

Graph Type	$\|\mathcal{S}\| = 20,\ \|\mathcal{A}\| = 5$	
	MSE	Accuracy
Star	2.22 ± 4.4	100.0 ± 0.0
Caveman	1.66 ± 1.1	98.1 ± 2.7
Caterpillar	1.33 ± 1.4	97.6 ± 3.2
Lobster	1.44 ± 2.5	97.4 ± 4.0
Tree	1.40 ± 1.5	94.5 ± 5.0
Grid	0.66 ± 0.7	94.1 ± 6.1
Ladder	0.68 ± 0.7	90.1 ± 5.9
Line	1.02 ± 1.0	90.8 ± 6.6

Table 4.1.2: These graphs have out-of-distribution graph structures but the same size as the train graphs. The overall MSE and Accuracy are slightly weakened but still satisfactory on both dense graphs (Star, Caveman) and sparse graphs (the rest).

(iii) Generalisation ability and scalability: on larger graph sizes

Graph Type	$\lvert\mathcal{S}\rvert = 50$, $\lvert\mathcal{A}\rvert = 10$		$\lvert\mathcal{S}\rvert = 100$, $\lvert\mathcal{A}\rvert = 20$	
	MSE	Accuracy	MSE	Accuracy
Erdős-Rényi	0.49 ± 0.2	98.8 ± 1.5	1.96 ± 0.2	99.4 ± 0.8
Barabási-Albert	0.48 ± 0.3	99.1 ± 1.3	1.98 ± 0.2	99.3 ± 0.9
Star	1.12 ± 1.1	99.9 ± 0.4	2.07 ± 0.8	100.0 ± 0.2
Caveman	0.62 ± 0.2	98.0 ± 2.1	1.98 ± 0.2	97.5 ± 1.3
Caterpillar	0.71 ± 0.3	94.0 ± 3.7	1.93 ± 0.2	96.7 ± 1.5
Lobster	0.76 ± 0.3	91.2 ± 4.5	1.96 ± 0.2	94.6 ± 2.0
Tree	0.71 ± 0.3	95.2 ± 3.0	2.00 ± 0.3	94.2 ± 2.2
Grid	0.58 ± 0.3	92.4 ± 4.0	2.02 ± 0.2	91.2 ± 3.0
Ladder	0.70 ± 0.2	92.2 ± 3.8	2.05 ± 0.2	90.8 ± 3.1
Line	0.84 ± 0.3	90.0 ± 4.3	2.10 ± 0.2	87.6 ± 3.5

Table 4.1.3: These graphs have out-of-distribution (larger) sizes from the train graphs. The error increases moderately as the graph expands, but the overall performance remains resilient.

We can see the GNN Executor can predict the policy and state-value function very **closely** to the value iteration outputs. This supports the algorithm alignment framework in suggesting that the close alignment between GNN and value iteration gives good predictability. Furthermore, by testing the GNN Executor on out-of-distribution graph types and graph sizes, we can see it also has good **generalisation ability**, demonstrating the power of Neural Algorithmic Reasoning. The ability to train on small-sized graphs also suggests good **scalability** and **efficiency**.

4.2 XLVIN on discrete control

XLVIN uses the above evaluated GNN Executor as a core processor to facilitate RL decision-making. It is also known to have good sample efficiency, tackling the limitation that most Deep RL agents suffer from. In order to evaluate these claims of XLVINs on discrete control problems, I propose the following questions:

Qn. How well can XLVINs perform in discrete control environments?

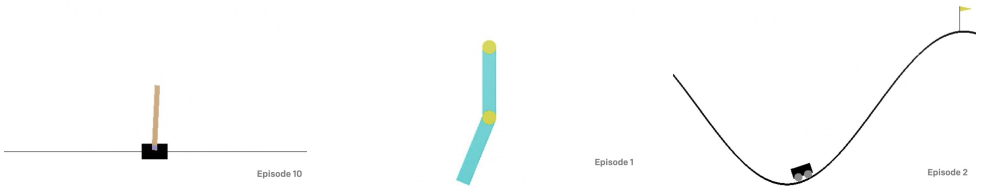
- *Can XLVINs outperform PPO, which is a state-of-the-art on-policy optimisation algorithm, on discrete control problems? (4.2.1)*
- *Is the GNN Executor learning meaningful state-value function predictions? (4.2.2)*
- *Can XLVINs generalise onto more complicated environments of the same type? (4.2.3)*

Models under evaluation:

- PPO Baseline: The baseline agent is implemented as a PPO agent that consists of `Encoder` and `Policy/Value` functions.
- XLVIN-R: On the basis of PPO Baseline, it has additional `Transition` and `Executor` functions. The GNN Executor is pre-trained using 500 random Erdős-Rényi graphs with size $(\lvert\mathcal{S}\rvert = 20, \lvert\mathcal{A}\rvert = 8)$.
- XLVIN-B: It differs from XLVIN-R in the way that the GNN Executor is pre-trained with 500 random binary trees defined as: nodes closer to the root have reward 1, while the nodes

farther away have reward 0. This type of binary tree discourages a monotone pursuit in a single direction, imitating the bi-directional control for the environments below.

Environment choices:



(i) CartPole-v1 (ii) Acrobot-v1 (iii) MountainCar-v0

Figure 4.2.1: Illustration of the selected discrete control environments.

The above models were evaluated on three environments (Figure 4.2.1) from OpenAI Gym [23]'s Classic Control suite. A detailed description and discussion of the difficulties for each environment can be found in Appendix A.4. In short, the set covers both dense and sparse rewards, and it poses a challenge to the agent in balancing between exploration and exploitation.

4.2.1 Performance, Stability, and Data Efficiency

Performance: The most commonly used evaluation metric in RL literature is the average episodic reward, which I aggregated over 100 test episodes. It directly measures how well the goal is met, thus the agent's performance.

Stability: RL agents are sensitive to the stochasticity of environments. I chose to calculate the standard deviation (std) of the average episodic rewards across 5 seeds to reflect stability.

Data efficiency: In order to evaluate whether XLVIN can produce better policy with fewer training samples, I compared the agents' performances under a data shortage condition: training with only 10 episodes for CartPole-v1, and 100 episodes for the other two environments.

Discussion of results:

Model	**CartPole-v1**	**Acrobot-v1**	**MountainCar-v0**
PPO Baseline	94.1 ± 22.2	-479.1 ± 20.9	-200.0 ± 0.0
XLVIN-R	**199.6** ± 0.4	-265.5 ± 64.6	-182.3 ± 16.0
XLVIN-B	186.3 ± 6.85	**-224.7** ± 68.9	**-162.2** ± 18.3

Table 4.2.1: Mean episodic rewards using PPO Baseline and two variants of XLVIN agents. The best performing model for each environment is highlighted in bold.

As seen in Table 4.2.1, both variants of XLVIN agents consistently demonstrate better **performance** over PPO Baseline, with higher average episodic rewards in all three environments. XLVIN-R has smaller variances, suggesting better **stability** than XLVIN-B. Furthermore, all three environments are conducted with a small number of train episodes, showing how XLVIN agents can learn in a **low-data regime**. This suggests that combining value iteration subroutines into an RL agent helps in producing better policy with good sample efficiency, addressing the data hunger problem suffered by most Deep RL agents.

4.2.2 Predictability

We have seen that the pre-trained GNN generalises well onto out-of-distribution data in Section 4.1. However, we also want to evaluate if it is still making meaningful predictions when plugged into XLVIN, especially when the node and edge feature definitions are different due to the lack of MDP data.

We can further test two hypotheses: (i) whether it predicts state-value function $V^*(s)$ accurately, and (ii) whether the prediction is not due to `Encoder`'s effect.

Environment: The XLVIN-R model was tested in a self-constructed Maze environment (Section 3.1.2). The Maze environment was used because a fully tabulated MDP is needed to compute the ground-truth values from value iteration, which is intractable for OpenAI Gym's environments.

Set-up: The mazes were ranked according to difficulty levels, based on the length of the shortest path from start to goal. Therefore I chose to train in a curriculum manner: the model was trained with mazes from the lowest difficulty and only allowed to proceed to the next level if it solved more than 95% of the train mazes.

Evaluation metrics: To test the two hypotheses, for each model trained until each difficulty level, I took the state embedding before `Executor` (i.e. `Encoder` outputs $\vec{h}_s$), and the updated state embedding after `Executor` (i.e. `Executor` outputs $\vec{\mathcal{X}}_s$), then conducted an R-squared test to measure their goodness of fit with the ground-truth value function $V^*(s)$ computed from value iteration. A higher R-squared value indicates better fit, thus better prediction.

Discussion of results:

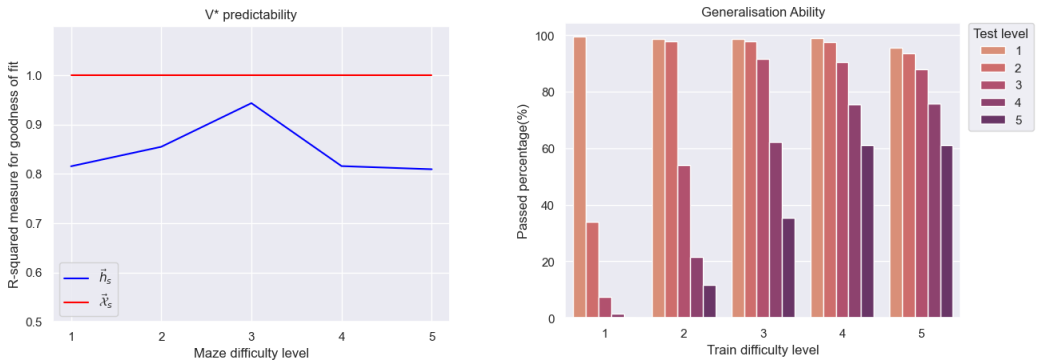


Figure 4.2.2: (a) R-squared measure on V^* predictability of Encoder and Executor outputs. (b) The success rate of agents by testing trained models on out-of-distribution test mazes.

From Table 4.2.2a, we can see `Encoder`'s output $\vec{h}_s$ only reaches around $R^2 \approx 0.85$ of fitting the ground-truth V^* values, while `Executor`'s output successfully raises the predictability to $R^2 \approx 1.0$. This indicates the accuracy of `Executor`'s outputs, as well as proving its contribution towards better predictions.

4.2.3 Generalisation ability

To see if the XLVIN model can generalise to problems different from the training environments. I took the models trained with mazes of each difficulty level and tested them on other difficulty levels. From Table 4.2.2b, we can see trained models at level=3 or 4 already can generalise reasonably well to more difficult maze levels, showing XLVINs' generalisation ability.

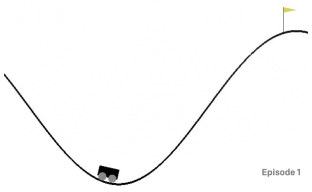
4.3 CNAP on continuous control

CNAP was proposed based on XLVIN for continuous control problems. CNAP discretises the continuous action space into discrete action bins. We want to explore the following questions:

Qn: How well can CNAPs perform in continuous control environments?

- *Can CNAPs outperform a discretised version of PPO Baseline on continuous control problems? (4.3.1)*
- *How many action bins should we discretise the continuous action space into? i.e. What is the optimal width of GNN? (4.3.2)*
- *How many steps of message-passing should we perform? i.e. What is the optimal depth of GNN? (4.3.3)*

Environment choice:



MountainCarContinous-v0

Figure 4.3.1: Illustration of the selected continuous control environment

To evaluate CNAPs on continuous control problems, I chose MountainCarContinuous-v0 (Figure 4.3.1) from OpenAI Gym's Classic Control suite. A detailed description is in Appendix A.4. In short, it allows the agent to specify a force magnitude from a continuous range. Furthermore, it puts larger penalties on larger forces, making it more challenging for the agent to balance the exploration-exploitation trade-off.

4.3.1 Performance, Stability, and Data Efficiency

Discussion of results:

Model	**MountainCarContinuous-v0**
PPO Baseline	-4.96 ± 1.24
CNAP-R	**63.41** ± 37.89
CNAP-B	55.73 ± 45.10

Table 4.3.1: Mean episodic rewards using PPO Baseline and two variants of CNAP agents. The best performing model is highlighted in bold. The continuous action space is discretised into 10 action bins. Both CNAP agents execute one step of message-passing.

As seen in Table 4.3.1, both two variants of the CNAP agents have significantly better **performances** than the PPO Baseline. One difficulty of this environment comes from how it penalises larger forces. The PPO Baseline gives an average episodic reward close to 0, which means it prefers not to do anything to avoid penalties and never manages to learn the successful policy. On the contrary, both CNAP agents have positive average rewards, suggesting that they can escape from being fooled by the temporary negative rewards and solve the problem in most

cases. Note the agent only receives a large positive reward when it succeeds, otherwise the overall rewards would be negative, explaining the larger variance for CNAPs. CNAP-R not only has the best performance but also demonstrates better **stability** than CNAP-B. Furthermore, the models are trained with only 100 episodes as before, so it suggests CNAPs retain the good **data efficiency** from XLVINs.

4.3.2 Effect of GNN width (Extension)

We made one crucial choice on the number of action bins when discretising the continuous action space. The number of action bins defines the size of the discretised action space and how much information to sacrifice from discretisation. It also defines the degree of the GNN nodes, which is the width of the GNN. It is natural to ask how this choice would impact CNAP's performance on continuous control problems.

Model	Action Bins	**MountainCarContinuous-v0**
PPO	5	-2.16 ± 1.25
	10	-4.96 ± 1.24
	15	-3.95 ± 0.77
CNAP-R	5	20.32 ± 53.13
	10	**63.41** ± 37.89
	15	26.21 ± 46.44
CNAP-B	5	29.46 ± 57.57
	10	55.73 ± 45.10
	15	22.79 ± 41.24

Table 4.3.2: Mean episodic rewards using PPO Baseline and CNAP agents with varied number of action bins. Both CNAP models perform one message-passing step.

As seen in Table 4.3.2, discretising into 10 action bins gives the best performance. This shows the importance of an appropriate action space size to give better performances. One possible explanation is that a smaller number of 5 action bins constrains the action choices too much, while a larger number of 10 action bins makes the policy more difficult to learn.

4.3.3 Effect of GNN depth (Extension)

Another choice we made was how many message-passing steps the GNN should perform, which is also the number of value iteration updates to simulate. The number of GNN steps also defines the depth of the graph constructed, i.e. how many layers of nodes to expand. Below, I explored the effect of using different numbers of GNN steps on the performance of CNAPs.

Model	GNN Steps	**MountainCarContinuous-v0**
CNAP-R	1	**63.41** ± 37.89
	2	34.49 ± 47.77
	3	43.61 ± 46.16
CNAP-B	1	55.73 ± 45.10
	2	46.93 ± 44.13
	3	40.58 ± 48.20

Table 4.3.3: Mean episodic rewards using CNAP agents with varied number of message-passing steps. All models discretise into 10 action bins.

As seen in Table 4.3.3, the best performance is given by using only one message-passing step, while increasing the number of steps weakens the performance. There are several possible explanations for this observation. One possibility is that a deeper GNN graph implies a larger size, which may require more training data. Similarly, an increase in depth relies on using the `Transition` function multiple times, which is also a function to be trained; furthermore, imprecision in `Transition` may add up as the graph goes deeper, leading to inappropriate next state embeddings. Lastly, the oversmoothness is an infamous side-effect of graph convolutions which exacerbates with the depth of a GNN [36]: Recall that message-passing uses an aggregate operation to gather messages from connected neighbours, which is a sum operator in our case. The aggregator may overly smooth the node features as more message-passing steps are performed, making nodes less distinguishable from each other, thus leading to suboptimal predictions.

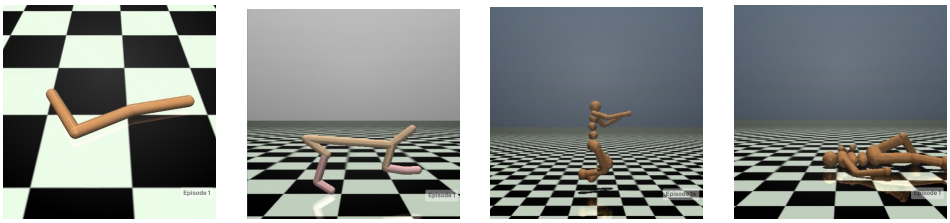
4.4 CNAP on complex continuous control (Extension)

While MountainCarContinuous-v0 only has one-dimensional action space, we want to evaluate whether CNAP can handle complex continuous control environments that go beyond a single dimension. CNAPs were extended with a factorised joint policy and a neighbour sampling method. Four possible methods for sampling were proposed in Section 3.4.2.2: (1) Manual-Gaussian (2) Learnt-Gaussian (3) Reuse-Policy (4) Learnt-Sampling. We want to explore the following questions:

Qn: How well can CNAPs perform in highly complex continuous control environments?

- *Can CNAPs outperform PPO Baseline in more complex environments with the proposed extensions? (4.4.1)*
- *How much performance do we lose from only sampling a subset of actions to expand rather than expanding all possible actions? (4.4.1.(i))*
- *Which neighbour sampling method works the best? (4.4.1)*
- *Can we interpret the results? (4.4.2)*

4.4.1 Scalability



(i) Swimmer-v2 (ii) Halfcheetah-v2 (iii) Humanoid-v2 (iv) Humanoid Standup-v2

Figure 4.4.1: Illustration of the selected MuJoCo environments

Environment choices: To evaluate CNAPs' ability in environments with complex dynamics, I selected four environments (Figure 4.4.1) from OpenAI Gym's MuJoCo suite [23][37] with increasing complexity. A detailed description can be found in Appendix A.4. In short, MuJoCo environments have much higher dimensions in both their observation and action spaces. They present a series of physical simulation and robotics control tasks closer to real-world domains.

Models under evaluation: Only CNAP-R (GNN trained with Erdős-Rényi graphs) was evaluated against PPO Baseline. CNAP-B (trained with binary trees) was no longer tested as it was proposed to suit environments with bi-directional control. Here, the environments have more complex dynamics than only two directions.

Evaluation metrics: MuJoCo environments converge slowly and take a lot more timesteps to learn. It is a common practice in RL literature to draw the learning curve of the training process. A learning curve means plotting the average episodic rewards against the number of training timesteps. Therefore a higher mean episodic reward indicates better **performance**, a smaller variance means higher **stability**, and a faster convergence suggests better **data efficiency**. I chose to aggregate the mean episodic rewards over 100 episodes, and repeat with 5 seeds.

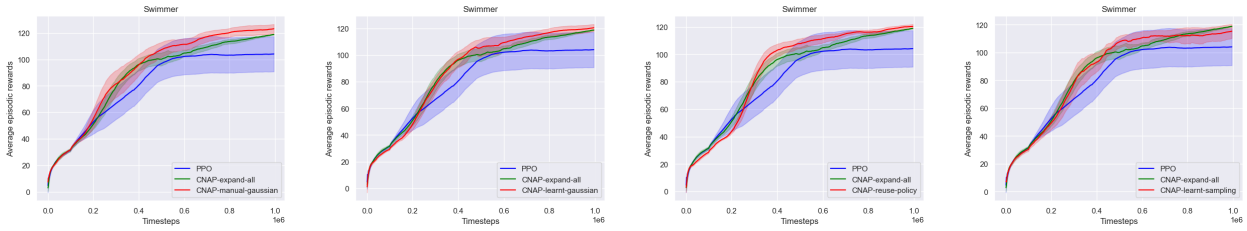
Suite	Environment	Observation space dimension	Action space dimension
Classic Control	MountainCar Continuous-v0	2	1
MuJoCo	Swimmer-v2	8	2
	HalfCheetah-v2	17	6
	Humanoid-v2	376	17
	HumanoidStandup-v2	376	17

Table 4.4.1: Comparison of environment complexity (dimensions of observation and action spaces) between MuJoCo suite and Classic Control suite.

To evaluate CNAP's scalability, below I present the evaluation in the order of increasing environment complexity, as illustrated in Table 4.4.1.

(i) Lower dimensional environment: Swimmer

I first tested the models on a relatively lower-dimensional environment: Swimmer-v2, where the action dimension is 2. This means it is still tractable to compare with a "benchmarking" CNAP that expands all possible actions as neighbours. For example, when the number of action bins[1] is 11, there are $11^2 = 121$ actions.



(a) Manual-Gaussian (b) Learnt-Gaussian (c) Reuse-Policy (d) Learnt-Sampling

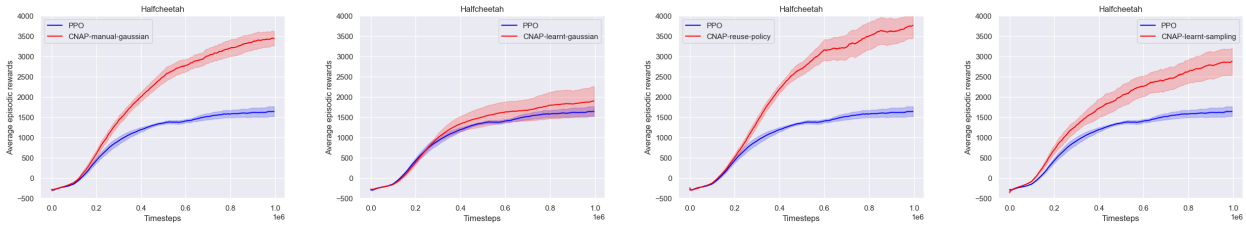
Figure 4.4.2: Learning curves of CNAP with expanding all actions (green), CNAP with neighbour sampling method (red), and PPO baseline (blue) in Swimmer using different sampling methods. The mean is shown by the line, and the standard deviation is illustrated by the shadow.

[1] According to previous work [33], the best performance on MuJoCo environments is obtained when $7 \leq N \leq 15$, where N is the number of action bins. I chose to take $N = 11$ here for all the experiments on MuJoCo tasks.

Results: As seen in Figure 4.4.2, all CNAPs outperformed the PPO Baseline. The four sampling methods all produced comparable performances as the "benchmarking" expanding-all-actions version, showing the effectiveness of the proposed sampling methods. Furthermore, Manual-Gaussian and Reuse-Policy performed better than Expand-All, suggesting that an appropriate selection of actions may filter out lower quality ones, producing better state-value function prediction and consequently better policy.

(ii) Higher dimensional environment: HalfCheetah

I then moved on to an environment with 6-dimensional action space, Halfcheetah. Here, an enumeration over the discretised action space becomes impossible. For example, 11 action bins in 6 dimensions means 11^6 actions.



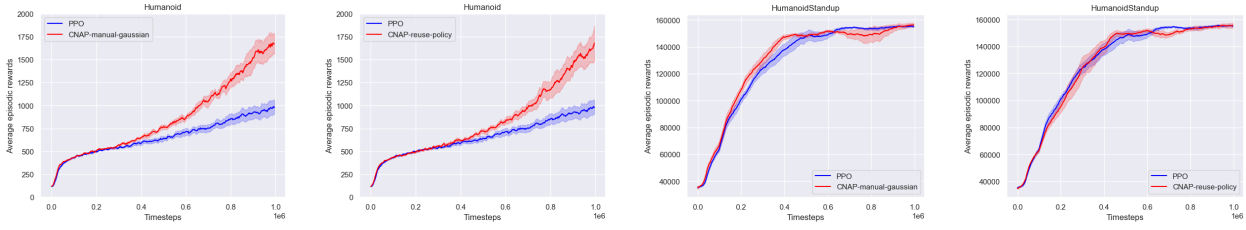
(a) Manual-Gaussian (b) Learnt-Gaussian (c) Reuse-Policy (d) Learnt-Sampling

Figure 4.4.3: Learning curves of CNAP (red) and PPO baseline (blue) in Halfcheetah, using different sampling methods.

Results: As seen in Figure 4.4.3, CNAPs outperform the PPO Baseline from the increased complexity in all cases. Again, Manual-Gaussian and Reuse-Policy are the most promising sampling strategies. This points to the importance of a well-chosen Gaussian distribution and the power of parameter reuse.

(iii) Even higher dimensional environments: Humanoid, HumanoidStandup

Finally, I tested CNAPs with Manual-Gaussian and Reuse-Policy on two environments with 17 action dimensions, Humaoid and HumanoidStandup. The robot is upgraded from a 2D halfcheetah to a 3D humanoid, which is intrinsically more difficult yet exhibits more potential in mapping onto real-world applications.



(a1) Manual-Gaussian (a2) Reuse-Policy (b1) Manual-Gaussian (b2) Reuse-Policy

Figure 4.4.4: Learning curves of CNAP (red) and PPO baseline (blue) in Humanoid (a) and HumanoidStandup (b), using Manual-Gaussian and Reuse-Policy.

Results: As we can see from Figure 4.4.4, the gain from CNAPs with sampling is also significant in Humanoid. However, in HumanoidStandup, we note that although Manual-Gaussian has a slight improvement over PPO Baseline, Reuse-Policy does not display much benefit. This brings us to take a closer look at the interpretation of results.

4.4.2 Interpretation Analysis

As discussed in Section 3.1, a `VideoRecorderWrapper` was used to schedule a video recording routine to capture the interaction of the agent with the environment. To interpret the learning curves, I chose to look at the selected frames at equal time intervals from one episode after the last training iteration by CNAP (Manual-Gaussian) and PPO Baseline, respectively.

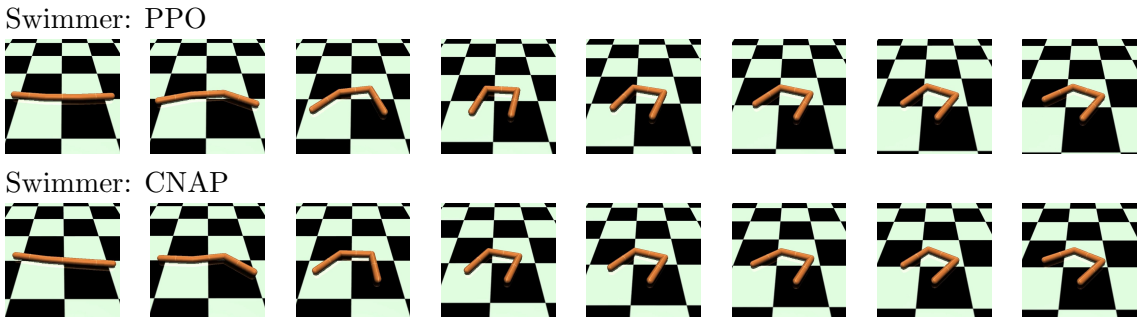


Figure 4.4.5: Selected frames of two agents in Swimmer

As seen in Figure 4.4.5, CNAP can fold itself slightly faster than PPO Baseline in this episode, and it also swims a little bit more quickly.

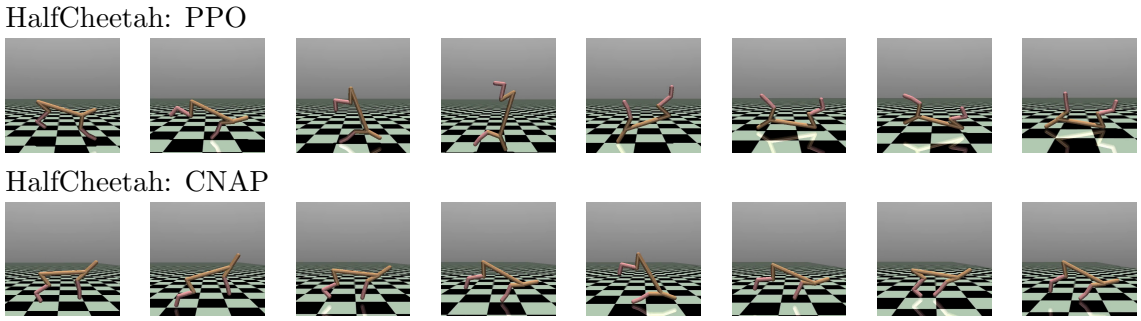


Figure 4.4.6: Selected frames of two agents in HalfCheetah

From Figure 4.4.6, we can see the agent instructed by PPO Baseline falls over quickly and never manages to turn it back. However, CNAP's agent can balance well and keeps running forward. This explains why CNAP has much higher average episodic rewards than PPO Baseline in Figure 4.4.3.

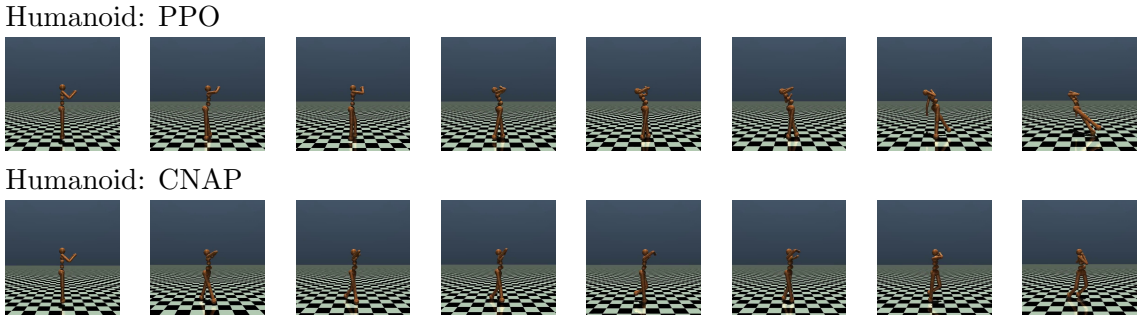
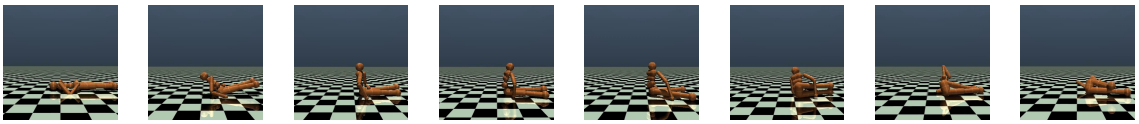


Figure 4.4.7: Selected frames of two agents in Humanoid

Similarly, in Figure 4.4.7, PPO Baseline's humanoid stays stationary and loses balance quickly, while CNAP's humanoid can walk forward in small steps. This aligns with our results in Figure 4.4.4 where the gain from CNAP is significant.

HumanoidStandup: PPO



HumanoidStandup: CNAP

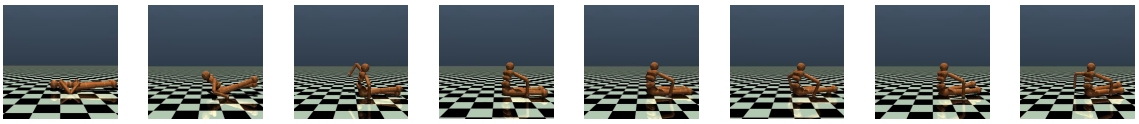


Figure 4.4.8: Selected frames of two agents in HumanoidStandup

Then we notice that although in Figure 4.4.4's HumanoidStandup task, the quantitative performances between PPO Baseline and CNAP are similar, Figure 4.4.8 reveals some different results. Both agents fail to stand up, explaining why the episodic rewards are similar numerically. However, the PPO Baseline agent loses balance and falls back to the ground while the CNAP agent remains sitting, trying to get up. We therefore can still say the CNAP qualitatively performs better in this example.

Discussion: The interpretation from video captures provides a qualitative explanation of the numerical results we obtained previously and reveals some unseen information. We can see CNAPs seem to benefit more as the complexity of the environment rises, which PPO Baselines can no longer solve. These are promising results as complexity has always been a challenge to RL agents, and the results prove the effectiveness of the proposed extensions in CNAPs. It also unveils the potential of CNAPs to be scaled up for more complex real-world RL applications.

4.5 Success Criteria

The project has met all Success Criteria of the core project, as well as all the Extensions. Based on the evaluation results above, I now demonstrate how the main deliverables have been achieved.

4.5.1 Core Project

- **GNN Executor:** I successfully implemented a message-passing GNN that simulates value iteration algorithm. I generated a set of synthetic graph-structured data for training and testing. The trained GNN has demonstrated high accuracy in mimicking value iteration and can generalise well onto out-of-distribution data (4.1).
- **XLVINs:** I set up the interfaces for a range of environments and constructed a pipeline for training and testing. I also built a competitive baseline agent using the PPO algorithm. I then implemented two variants of XLVIN models, which both demonstrated superior performances to the PPO Baseline on a set of discrete control environments (4.2.1). The results showed that XLVINs could learn well even within a low data regime, aligning with the original paper's results of being sample efficient. Then I further evaluated the predictability (4.2.2) and generalisation ability (4.2.3) of the plugged-in GNN Executor with a self-constructed Maze environment, demonstrating its effectiveness as a core processor for XLVINs.
- **CNAPs:** I extended XLVINs into CNAPs by discretising the continuous action space to allow the handling of continuous control. CNAPs outperformed PPO Baseline in

the selected continuous environment, demonstrating how CNAPs inherited the sample efficiency from XLVINs while extending the solvable problem scope (4.3.1).

4.5.2 Extensions

- **Performance with GNN size:** I explored the performance of CNAPs with varied GNN sizes, taking a deeper look at the relationship between model effectiveness and architectural set-up (4.3.2; 4.3.3). I also proposed several explanations for the empirical results.
- **Scale to multi-dimensional action space:** I identified two architectural bottlenecks in XLVINs and then proposed to use a factorised joint policy to deal with higher dimensional action space.
- **Scale to highly complex action space:** I then proposed to use a neighbour sampling method to tackle the second bottleneck, with four different solutions. I evaluated CNAPs with the two extensions on a set of increasingly complex continuous environments, where CNAPs showed significant improvement over the PPO Baseline (4.4.1). I also provided an interpretation analysis of the numerical results (4.4.2). The results positively showed that CNAPs could perform well in high-dimensional continuous control problems with complex dynamics. This contributes to filling the gap between a theoretical model and real-world reinforcement learning applications.
- **Open-source as a package:** Finally, since XLVIN had not released its code by the time of my dissertation, I open-sourced my implementation together with my extensions as a package using `setuptools` to benefit the wider research community with transparency and accessibility.

Chapter 5

Conclusions

In this chapter, I conclude with success achieved by this project (5.1), and discuss my reflection and lessons learnt (5.2). Finally, I point directions for possible future work (5.3).

5.1 Achievements

The project was a success, meeting all Success Criteria and Extensions. I successfully implemented XLVINs despite the numerous hyperparameters and stochasticity in environments that make reproducing in RL difficult. Then I proposed CNAPs, which broke the complexity constraints faced by XLVINs and demonstrated CNAPs' effectiveness in a series of complex continuous control tasks. I successfully expanded the application scope of such an RL agent empowered by algorithmic reasoning. The ability to solve complex continuous tasks reveals CNAPs' potential in more real-world applications, such as autonomous driving and robotics simulation. Furthermore, CNAPs can produce better policy under low data settings. The sample efficiency makes CNAPs more desirable, especially for applications where data collection is difficult and costly, addressing the data hunger suffered by conventional Deep RL agents.

Apart from its interests in Deep RL, my project also demonstrate the power of Neural Algorithmic Reasoning, tackling the rigid input constraints in classical algorithms. Additionally, it proposes a novel setup to align discrete neural network processes with discrete algorithms under continuous inputs, and it shows we can incorporate appropriate techniques to address the architectural bottlenecks. These opened a new pathway in demonstrating how we can combine classical algorithms with the broader machine learning domain closer together.

5.2 Lessons Learnt

Given the novelty of the topics involved, I equipped myself with most theories by reading relevant research papers, where many useful ones came out even during the project. Given the exploratory nature of extensions, it proved to be an essential skill to learn how to research for a broader and deeper understanding.

I also honed my project management skill which required careful planning and quick adaptation due to the tight time constraint and uncertainty in extensions. Open-sourcing the implementation also trained me to maintain good software engineering techniques and clear documentation throughout the project.

Upon reflection, I found it tedious to fine-tune the numerous model hyperparameters, given that each set of experiments can take days to complete. The problem was exacerbated by the difficulty in explaining Deep RL models. I would adopt a more systematic and deterministic approach for future projects.

5.3 Future Work

There are several possible future directions that I propose as follows, which were not explored due to the time constraint of this project:

- **Other design choices:** I used a factorised joint policy from [33] to deal with the combinatorial explosion resulting from discretisation. However, there are other possible choices, such as the sequential policy model proposed by [31] which may bring other benefits.
- **Ordinal relationship in discretisation:** One drawback of the discretisation technique I used was how the continuous actions are now treated as independent discrete action bins, losing the ordinal relationship. Many previous works dedicated to this problem, such as [38] which introduces a stick-breaking likelihood to incorporate ordinal information, may also enhance the current models.
- **Other in-depth interpretation analysis:** I chose to interpret the learning curves by looking at video recordings, but this approach lacks a quantitative interpretation to precisely align with the numerical results. A more rigorous interpretability analysis on model explanation, outcome explanation, and model inspection [39] will be helpful, especially before employing them in safety-critical applications. Previous work such as [40], where they use the produced trajectory to train a Soft Decision Tree [41] and interpret the learnt policy, can be a good starting point.
- **Other quantitative evaluation methods:** Rishabh et al. (2021) [42] recently argued that due to RL agents' sensitivity towards stochasticity, the current evaluation methods should be strengthened. They proposed three evaluation methods for a more rigorous analysis across runs and tasks. These will be very helpful in further evaluating CNAPs with more credibility and in broader domains. Furthermore, other on-policy optimisation algorithms that are more capable of solving continuous control tasks than PPO can also be used as baseline models.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN: 978-0-262-19398-6.

[2] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. "An Introduction to Deep Reinforcement Learning". In: *Found. Trends Mach. Learn.* 11.3-4 (2018), pp. 219–354. DOI: 10.1561/2200000071.

[3] *AWS DeepRacer Documentation*. https://docs.aws.amazon.com/deepracer.

[4] Jiwei Li, Will Monroe, Alan Ritter, Dan Jurafsky, Michel Galley, and Jianfeng Gao. "Deep Reinforcement Learning for Dialogue Generation". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. The Association for Computational Linguistics, 2016, pp. 1192–1202. DOI: 10.18653/v1/d16-1127.

[5] Aniruddh Raghu, Matthieu Komorowski, Imran Ahmed, Leo Celi, Peter Szolovits, and Marzyeh Ghassemi. *Deep Reinforcement Learning for Sepsis Treatment*. 2017. DOI: 10.48550/ARXIV.1711.09602.

[6] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4.

[7] Petar Veličković and Charles Blundell. "Neural algorithmic reasoning". In: *Patterns* 2.7 (July 2021), p. 100273. DOI: 10.1016/j.patter.2021.100273.

[8] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. "What Can Neural Networks Reason About?" In: *8th International Conference on Learning Representations*. 2020.

[9] Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. "Neural Execution of Graph Algorithms". In: *8th International Conference on Learning Representations*. 2020.

[10] Andrew Dudzik and Petar Velickovic. "Graph Neural Networks are Dynamic Programmers". In: *CoRR* abs/2203.15544 (2022). DOI: 10.48550/arXiv.2203.15544.

[11] Andreea Deac, Petar Velickovic, Ognjen Milinkovic, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolic. "Neural Algorithmic Reasoners are Implicit Planners". In: *Advances in Neural Information Processing Systems 34*. 2021, pp. 15529–15542.

[12] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. "End-to-End Training of Deep Visuomotor Policies". In: *J. Mach. Learn. Res.* 17 (2016), 39:1–39:40.

[13] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, and Diego et al. de las Casas. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602.7897 (2022), pp. 414–419. DOI: 10.1038/s41586-021-04301-9.

[14] *GroundedML Workshop, the Tenth International Conference on Learning Representations (ICLR 2022)*. https://sites.google.com/view/groundedml2022.

[15] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN: 9780486428093.

[16] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[17] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. "Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges". In: *CoRR* abs/2104.13478 (2021). arXiv: 2104.13478.

[18] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. PMLR, 2017, pp. 1263–1272.

[19] Andreea Deac, Pierre-Luc Bacon, and Jian Tang. "Graph neural induction of value iteration". In: *CoRR* abs/2009.12604 (2020). arXiv: 2009.12604.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347.

[21] David Saad. *On-Line Learning in Neural Networks*. Publications of the Newton Institute. Cambridge University Press, 1999. DOI: 10.1017/CBO9780511569920.

[22] Ilya Kostrikov. *PyTorch Implementations of Reinforcement Learning Algorithms*. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail. 2018.

[23] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[24] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[25] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. "GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. 2018, pp. 5694–5703.

[26] Jiaxuan You, Rex Ying, and Jure Leskovec. "Position-aware Graph Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. PMLR, 2019, pp. 7134–7143.

[27] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning*. Omnipress, 2010, pp. 807–814.

[28] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *CoRR* abs/1607.06450 (2016). arXiv: 1607.06450.

[29] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. "Translating Embeddings for Modeling Multi-relational Data". In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013.

[30] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. "Supervised Contrastive Learning". In: *Advances in Neural Information Processing Systems 33*. 2020.

[31] Luke Metz, Julian Ibarz, Navdeep Jaitly, and James Davidson. "Discrete Sequential Prediction of Continuous Actions for Deep RL". In: *CoRR* abs/1705.05035 (2017). arXiv: 1705.05035.

[32] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. "Action Branching Architectures for Deep Reinforcement Learning". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 4131–4138.

[33] Yunhao Tang and Shipra Agrawal. "Discretizing Continuous Action Space for On-Policy Optimization". In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2020, pp. 5981–5988.

[34] Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. "Learning and Planning in Complex Action Spaces". In: *Proceedings of the 38th International Conference on Machine Learning*. Vol. 139. PMLR, 2021, pp. 4476–4486.

[35] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations*. 2015.

[36] Qimai Li, Zhichao Han, and Xiao-Ming Wu. "Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 3538–3545.

[37] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[38] Mohammad Khan, Shakir Mohamed, Benjamin Marlin, and Kevin Murphy. "A Stick-Breaking Likelihood for Categorical Data Analysis with Latent Gaussian Models". In: *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*. Vol. 22. PMLR, 2012, pp. 610–618.

[39] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. "A Survey of Methods for Explaining Black Box Models". In: *ACM Comput. Surv.* 51.5 (2019), 93:1–93:42. DOI: 10.1145/3236009.

[40] Youri Coppens, Kyriakos Efthymiadis, Tom Lenaerts, and Ann Nowé. "Distilling Deep Reinforcement Learning Policies in Soft Decision Trees". In: *the 28th International Joint Conference on Artificial Intelligence*. 2019.

[41] Nicholas Frosst and Geoffrey E. Hinton. "Distilling a Neural Network Into a Soft Decision Tree". In: *CoRR* abs/1711.09784 (2017). arXiv: 1711.09784.

[42] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C. Courville, and Marc G. Bellemare. "Deep Reinforcement Learning at the Edge of the Statistical Precipice". In: *Advances in Neural Information Processing Systems 34*. 2021, pp. 29304–29320.

[43] Paul Erdös and Alfréd Rényi. "On Random Graphs. I". In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[44] Albert-Laszlo Barabasi and Reka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[45] Duncan J. Watts. "Networks, Dynamics, and the Small-World Phenomenon". In: *The American Journal of Sociology* 105.2 (1999), pp. 493–527. DOI: 10.2307/2991086.

[46] Kenjiro Ogawa, Yuhei Shiraki, Satoshi Tagusari, and Morimasa Tsuchiya. "On strict-double-bound numbers of caterpillars". In: *Discret. Math. Algorithms Appl.* 7.3 (2015), 1550038:1–1550038:7. DOI: 10.1142/S179383091550038X.

[47] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

[48] Richard S. Sutton. "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding". In: *Advances in Neural Information Processing Systems 8*. MIT Press, 1995, pp. 1038–1044.

[49] Andrew William Moore. *Efficient Memory-based Learning for Robot Control*. Tech. rep. Computer Laboratory, University of Cambridge, 1990.

[50] Rémi Coulom. "Reinforcement Learning Using Neural Networks, with Applications to Motor Control. (Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur)". PhD thesis. Grenoble Institute of Technology, France, 2002.

[51] Yuval Tassa, Tom Erez, and Emanuel Todorov. "Synthesis and stabilization of complex behaviors through online trajectory optimization". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 4906–4913. DOI: 10.1109/IROS.2012.6386025.

Appendix A

Additional information

A.1 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) [20] is an on-policy optimisation algorithm based on policy gradients. It performs a local search for the optimal policy that maximises the overall return. On-policy means it uses the same policy to generate data and compute the loss, while off-policy uses a different policy for data generation. PPO's loss function consists of three components:

$$L_{\text{PPO}}(\theta) = L^{\text{CLIP+VF+}S}(\theta) = \mathbb{E}[L^{\text{CLIP}}(\theta) - c_{vf}L^{\text{VF}}(\theta) + c_s S[\pi_\theta](s)] \tag{A.1.1}$$

where θ is the policy parameter, c_{vf} and c_s are coefficients.

(i) Clipped surrogate objective

The main component of PPO's loss function is a clipped surrogate objective:

$$L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{A.1.2}$$

where $r_t(\theta)$ is the ratio between the current policy π_θ and previous policy $\pi_{\theta_{old}}$,

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{A.1.3}$$

and $\hat{A}_t$ is the advantage function that measures how desirable an action is in the current state, computed by Generalised Advantage Estimation (GAE):

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{A.1.4}$$

where γ is the discount and λ is a smoothing parameter,

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{A.1.5}$$

We can interpret the surrogate objective $r_t(\theta)\hat{A}_t$ as finding the best policy π_θ that optimises an estimate of the expected cumulative rewards. The clipping operation keeps the new policy close to the old one so that it counters the large gradient variance problem.

(ii) Squared-error loss

PPO's loss function also includes the squared error of value function:

$$L_t^{\text{VF}}(\theta) = (V_\theta(s_t) - V_t^{target})^2 \tag{A.1.6}$$

where $V_\theta(s_t)$ is the state-value function under current policy θ, and V_t^{target} is the target value.

(iii) Entropy bonus

Lastly, the loss function also takes into account of an entropy bonus for sufficient exploration:

$$S[\pi_\theta](s_t) \tag{A.1.7}$$

The PPO loss objective is compatible with Stochastic Gradient Descent (SGD), and its simplicity has made it a default algorithm in OpenAI. It has been empirically found to be sample efficient while demonstrating good performance.

A.2 Pseudocode for Executor

`Executor` ($X : \mathbb{R}^k \to \mathbb{R}^k$) takes a state embedding $\vec{h}_s$ and constructs a graph where a pre-trained GNN can execute message-passing to imitate value iteration algorithm. The following pseudocode aids the explanation in Section 3.3.2 on how such a graph is constructed.

Algorithm 3 Executor

Input : State embedding $\vec{h}_{in}$ from Encoder
Output: Updated state embedding $\vec{\mathcal{X}}_s$
currentLayer = $[(\vec{h}_{in}, \_)]$;
for *i = 1,2,...,t* **do**
  for $(\vec{h}_s, \_)$ *in currentLayer* **do**
    for $a \in \mathcal{A}$ **do**
      $\vec{h}'_s = T(\vec{h}_s, a)$; `// node feature`
      $\vec{e}_{s' \to s}$=concatenate(a, γ) ; `// edge feature`
      $\mathcal{N}(\vec{h}_s)$.append$(\vec{h}'_s, \vec{e}_{s' \to s})$; `// neighbours of` $\vec{h}_s$
    end
  end
  currentLayer $\leftarrow \{\mathcal{N}(\vec{h}_s)\}$, for all $\vec{h}_s$ seen in *currentLayer*
end
`/* The graph` $G = (V, E)$ `is stored as:` */
/* $V = \{node\_feature\}$ */
/* $E = \{(sender, receiver, edge\_feature)\}$ */
`/* which are inputs to the pre-trained message-passing GNN to imitate value iteration behaviour, producing` $\vec{\mathcal{X}}_s$. */

A.3 List of graph types

The GNN Executor was evaluated on a list of different graph types in Section 4.1. Here is a description of each graph type used for evaluation.

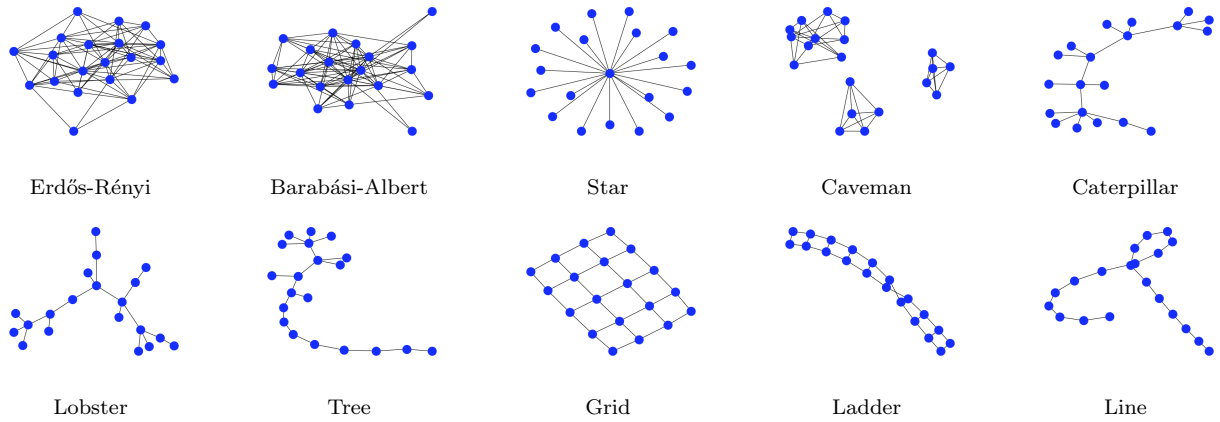


Figure A.3.1: Illustration of each graph type created with `networkx`

Erdős-Rényi [43]: A graph of n nodes is constructed by randomly connecting two nodes by an edge with probability p, independently from all other edges.

Barabási-Albert [44]: A graph of n nodes is grown by attaching new nodes with m edges following the Barabási-Albert preferential attachment model where more connected nodes will more likely to receive new links.

Star: A graph of $n + 1$ nodes is constructed by attaching n nodes to a centre node.

Caveman [45]: A graph is constructed by randomly forming l cliques with clique size k.

Caterpillar [46]: A graph is constructed from a backbone of size n, and randomly attaching m nodes to the backbone.

Lobster: A graph is constructed from a backbone of size n, and randomly connecting m nodes to the backbone, where the m nodes are further randomly prolonged by l nodes.

Tree: A tree of size n is constructed with a power law degree distribution. This means for a node with degree k, probability $p(k) \propto k^{-\gamma}$ where γ is a parameter setting the exponent.

Grid: A 2-dimensional grid of size $n \times m$.

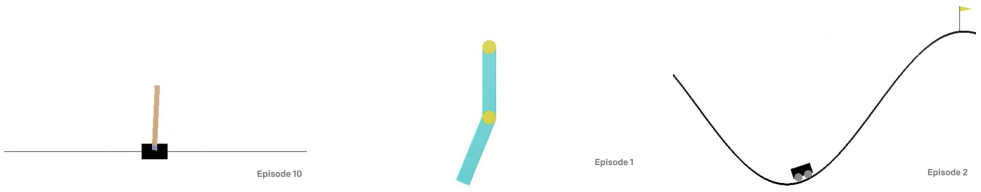
Ladder: A 2-dimensional grid of size $2 \times n$.

Line: A line consisting of n nodes.

A.4 List of environments

In Chapter 4, XLVINs and CNAPs were evaluated in a range of RL environments. Here is a description of each environment used.

A.4.1 Discrete Classic Control



CartPole-v1 Acrobot-v1 MountainCar-v0

Figure A.4.1: Illustration of discrete classic control environments from [23]

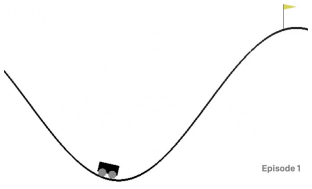
CartPole-v1 [47]: The goal is to maintain a pole loosely connected to a cart to stay upright as long as possible. The cart is placed on a frictionless surface. The agent can push the cart to the left or right at each timestep, and it receives a reward of +1 for each successful timestep. The episode terminates when the pole falls more than 15 degrees from vertical or if the cart moves more than 2.4 units from the starting position. It also ends on timeout at 200 timesteps.

Acrobot-v1 [48]: The system consists of two links connected via an actuated joint. The top link is attached firmly to the ceiling. The system starts vertical, and the goal is to swing the end of the lower link up to a certain height as quickly as possible. The agent can perform 3 possible actions: push left, no action, or push right. It receives a negative reward of -1 until success or until time-out at 500 timesteps.

MountainCar-v0 [49]: The car is initially positioned at the bottom of the valley. The agent aims to drive up the car to the right hill as quickly as possible by accumulating momentum from applying an acceleration to the left or right, or doing nothing. It receives a negative reward of -1 at each timestep until success or until time-out at 200 timesteps.

There are two notable challenges in these environments. Firstly, on dense and sparse rewards: CartPole-v0 rewards at each successful timestep, thus giving dense rewards. However, Acrobot-v1 and MountainCar-v0 only give large positive feedback when the goal is reached, which are sparse rewards. Sparse rewards are harder to learn as the agent is less guided in evaluating how valuable the actions are. Secondly, on exploration and exploitation: Acrobot-v1 and MountainCar-v0 penalise at each timestep for not yet reaching the goal. The agent thus should learn how to balance the trade-off between sufficient exploration of the environment and exploitation of the best action combinations.

A.4.2 Continuous Classic Control



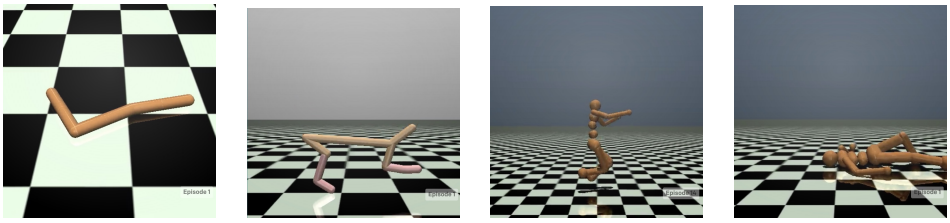
MountainCarContinous-v0

Figure A.4.2: Illustration of continuous classic control environments from [23]

MountainCarContinuous-v0 [49]: It extends from MountainCar-v0 in the way that it requires more precision by allowing the agent to specify the action's magnitude from a continuous range. For example, applying a left acceleration with magnitude 0.2 corresponds to action $a = 0.2$. The goal is to drive the car up to the right hill as quickly as possible, with minimum forces applied. Therefore, the agent receives a negative reward of $-0.1 * a^2$, relative to the magnitude of the force applied. The episode ends on success with a positive reward of +100 or on a timeout of 999 timesteps.

Besides the intrinsic difficulty from a discrete to continuous action space, the problem also poses a more significant penalty at larger forces. The penalty means the agent should additionally balance the trade-off between exploring a larger force while keeping the overall penalty low. For example, since a positive reward is only given if the goal is reached, an agent that does nothing until timeout gives better results than an agent that explores different action magnitudes but fails.

A.4.3 MuJoCo



Swimmer-v2 Halfcheetah-v2 Humanoid-v2 Humanoid Standup-v2

Figure A.4.3: Illustration of MuJoCo environments from [23][37]

Swimmer-v2 [50]: The system consists of a 3-link robot connected by two joints in a viscous fluid. The robot can twist its two joints, with the goal to swim forward as fast as possible.

HalfCheetah-v2: The robot is a 2-dimensional halfcheetah with two legs. The goal is to make the 2D robot run forward as fast as possible by actuating its six joints in the legs.

Humanoid-v2 [51]: The goal is to make a 3-dimensional two-legged robot move forward as fast as possible. The robot can operate its 17 joints all over its body to maintain balance and perform walking.

HumanoidStandup-v2: The robot is the same as in Humanoid-v2, but initially lying down on the ground. The goal is to operate its 17 joints to stand up as quickly as possible.

The MuJoCo environments all time out at 1000 timesteps. They offer a set of complex continuous control tasks that simulate real-life problems in robotics and animation. It can also test and validate control schemes before actual deployment into physical agents. The complexity of its dynamics and contact-rich behaviour needs the agents to scale up for computationally-intensive requirements.

A.5 List of hyperparmeters

A.5.1 GNN Executor

Name	Value	Description
lr	0.005	Learning rate of optimiser.
epsilon	0.0001	Epsilon value to determine convergence.
hidden_dimension	50	Dimension of the hidden layer.

Table A.5.1: List of hyperparameters for GNN Executor

A.5.2 Encoder/Transition

Name	Value	Description
state_embedding_dimension	50	Dimension of state embeddings in the latent space.
num_epochs	10	Number of epochs used for training.
batch_size	128	Batch size used for training.
hidden_dimension	64	Dimension of the hidden layer.

Table A.5.2: List of hyperparameters for `Encoder` and `Transition`

A.5.3 PPO

Name	Value	Description
lr	0.0003	Learning rate of optimiser.
gamma	0.99	Discount factor γ determining how much emphasis is put in the distant future rather than the immediate future.
clip_param	0.2	Clipping parameter ϵ controlling the degree of clipping.
value_loss_coef	0.5	Coefficient c_v for $L^{\mathrm{VF}}(\theta)$.
entropy_coef	0.01	Coefficient c_s for $S[\pi_\theta]$.
transe_loss_coef	0.001	Coefficient λ for L_{TransE}.
gae_lambda	0.95	Smoothing parameter in GAE's λ-return for stability.
max_grad_norm	0.5	Degree of clipping on norm of gradients.
ppo_epoch	5	Number of PPO training epochs.
num_minibatch	128	Number of minibatches for SGD.

Table A.5.3: List of hyperparameters for training with PPO

Appendix B

Project Proposal

Combining Classical Algorithms and Deep Reinforcement Learning Agents

2428G

18 Oct, 2021

Introduction

A Reinforcement Learning (RL) problem involves a learning agent that interacts with the environment to achieve an explicit goal. Its interaction with different domains has benefited many real-life applications, such as automated medical diagnosis and robotics manipulation. These problems usually require *planning*, which means we need to take possible future situations into account when we select actions to take.

Specifically, the agent can observe the *state* of the environment and learn to take *actions* to influence the state. The agent chooses which action to take depending on the state according to a *policy*. The environment responds with a *reward* signal, and the *goal* is to maximise the total reward the agent receives over the long run. A *value function* specifies how good the state is, that is, the total amount of reward an agent is expected to receive over time from that state.

Value iteration is a dynamic programming algorithm that can iteratively estimate the value function, from which we can obtain the optimal policy. Graph Neural Networks (GNNs) are well-suited to learn the value iteration function due to the alignment between value iteration and graph convolution rules. This has been proven to be effective in the previous work of XLVIN [11] which shows GNNs can model value iteration functions under supervised learning settings.

One limitation of XLVIN is that it only supports RL problems in discrete action space, but not continuous ones. Continuous space is more complex because there is an infinite number of feasible actions. However, there are many important RL problems whose action space is continuous in nature, especially in simulated or real-life system control tasks, such as navigation systems. Many branches of RL problems that involve continuous and dynamic activities fall under this category.

Another limitation of XLVIN is its computation complexity. Given a state, we need to construct a graph by recursively expanding on each state with all the possible actions that it can take, after which we can execute value iteration on the graph to estimate the state's value function. Thus, the size of the graph grows exponentially by the number of possible actions.

Description of project

In this project, I will reproduce XLVIN first by implementing its four core functions:

- Encoder function: A Multilayer Perceptron (MLP) encoder that translates state representation from the environment to a flat embedding in the latent space.
- Transition function: A Multilayer Percentron (MLP) with TransE [29] loss function that takes the state embedding and an action, and produces the resultant state embedding in the latent space.
- Executor function: A Graph Neural Network (GNN) that models the value iteration function [19], which takes a state and its immediate neighbours' embeddings to produce an updated state embedding.
- Actor/Tail functions: The actor function takes the outputs from above and produces a policy for a given state. The tail function takes the same inputs and produces the state-value function, which is the expected return from this state.

I will then run the agent on discrete tasks using CartPole, Acrobot, and MountainCar from OpenAI Gym [23].

- CartPole-v0: A cart is placed on a frictionless surface, with a pole attached to it by a joint. The goal is to prevent the pole from falling down by applying forces to the left or right to the cart. Reward is given at each timestep if the pole is maintained.
- Acrobot-v1: Two links are attached together by a joint, with the higher link attached to the ceiling. We can apply left or right forces to the lower joint. The goal is to swing the lower end of the lower link to reach a given height. Reward is given at each timestep based on whether this is achieved.
- MountainCar-v0: A car is positioned at the col of two mountains. It can drive back and forth to build up enough momentum, in order to reach the top of the mountain on the right. Reward is given when the goal is achieved, minus the squared sum of actions taken.

The three environments were selected to demonstrate the transferability of the agent. CartPole and Acrobot provide dense rewards where rewards are given at each timestep, while MountainCar provides sparse reward where reward is given only when car reaches the top of mountain. Furthermore, the penalty of taking actions in MountainCar also exposes an exploration challenge on whether to continue applying actions.

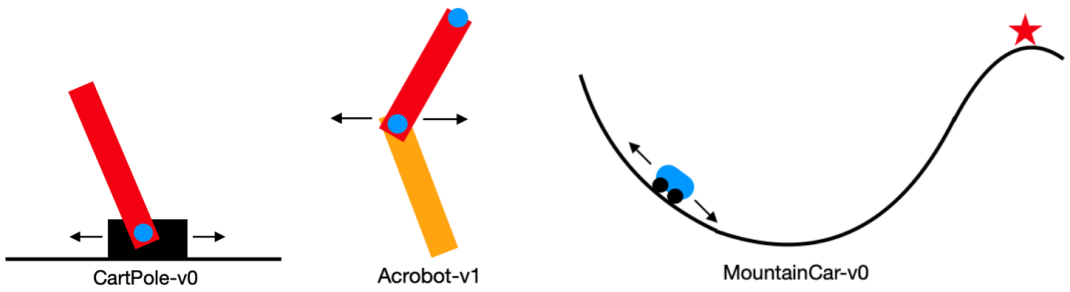


Figure B.0.1: Illustrations of the CartPole, Acrobot, and MountainCar environments

Then, since the current XLVIN only supports RL problems on discrete action spaces, I will extend XLVIN onto problems with continuous action space. I will leverage the results from [33] and primarily discretize the continuous action space into a finite set of atomic actions. A continuous XLVIN has not been implemented before and will extend its application scope to a much wider range as discussed in the Introduction. I will run the agent on the MountainCarContinuous-v0

from OpenAI Gym, where it also takes the force of driving into account, thus the action space is continuous. The reward is greater if less energy is required.

As one extension, I can run the agent on more advanced environments from MuJoCo [37], which contains a set of continuous control tasks that simulates tasks such as robot control and animation. One environment I can possibly use is Humanoid.

Another extension is to explore the performance of the agent with the number of atomic actions chosen when discretizing the continuous space.

One other extension is to improve the performance when constructing the graph. Currently, all states are expanded with each action, leading to exponential increase in state space. I can learn a policy or use non-parametric methods to selectively expand on some of the states instead.

Success criteria

- Construct the interface to run the environments
- Implement a baseline agent using Proximal Policy Optimization (PPO) [20]
- Pre-train the executor function of the agent
- Implement an agent that can run on discrete action space, and compare with the PPO baseline
- Implement an agent that can run on continuous action space, and compare with "discretized" PPO baseline

Possible Extensions

- Run the agent on more complicated continuous environments from MuJoCo
- Explore the performance of the agent with the number of atomic actions chosen when discretizing the continuous action space
- Implement a policy to selectively expand the states when constructing the graph
- Pack the implementations into an open source library

Evaluation

Executor:

One of the agent's components is a pre-trained executor is where GNN models value iteration algorithm. It can be tested on a set of synthetic graphs to evaluate how well it estimates the value functions.

Before the executor runs, an encoder function is used to map raw states to the latent space. To evaluate if the executor did meaningful improvement on predicting the value functions, we can compare the state embeddings before and after the executor to the ground truth values. Here, mazes will be used as the environment to test on.

Agent:

A model-free agent without transition and executor functions can be used as a baseline for evaluation. It will use Proximal Policy Optimisation (PPO) for parameter optimisation.

I will be using environments from OpenAI Gym for testing our agents. For discrete action space environments, the agent will be tested on CartPole-v0, Acrobot-v1, and MountainCar-v0. And for continuous action space environments, I will use MountainCarContinuous-v0.

Our agents will be evaluated against the baseline agent by tracking the scores obtained by the agents across a given number of episodes.

Starting Point

I have no experience in graph representation learning, however I have taken relevant courses in Machine Learning and Real-World Data, Data Science, and Artificial Intelligence from Computer Science Tripos Part IA and IB. I have limited exposure to PyTorch and have not used its relevant libraries to implement Graph Neural Networks. I have also never used OpenAI Gym before. I will learn to use the libraries and environments as I start to implement the model. I also had no knowledge in reinforcement learning, so I had to read various literature and the book *Reinforcement Learning - An Introduction book* by Andrew Barto and Richard S. Sutton [1] during the summer in order to understand the theoretical concepts behind XLVIN.

Timetable

Michaelmas Term:

1. **Week 2-3 (15/10/21-24/10/21)**:
   - Read up materials on RL.
   - Learn to use the libraries.
   - Interact with OpenAI Gym environments.

   Milestone: Acquire the foundations needed for the project.
2. **Week 4-5 (25/10/21-07/11/21)**:
   - Implement the interface for environments.
   - Implement encoder function with MLP.
   - Implement transition function with TransE.

   Milestone: Tested encoder and transition functions with the interface.
3. **Week 6-7 (08/11/21-21/11/21)**:
   - Implement GNN executor of value iteration.
   - Implement actor and tail functions with open-source PPO.

   Milestone: A baseline agent that can run on MountainCar.
4. **Week 8-9 (22/11/21-05/12/21)**:
   - Combine the functions and pretrain.
   - Evaluate the pretrained executor.

- Evaluate the agent on discrete problems.

Milestone: A pre-trained executor. An agent that can run on discrete action space. Evaluation results in the form of diagrams and tables.

Winter Holiday:

1. **Week 1-2 (06/12/21-19/12/21)**:
   - Implement the discretization of the continuous action space.

   Milestone: Discretization implemented with modifications to other functions.
2. **Week 3-4 (20/12/21-02/01/22)**:
   - Evaluate the agent on continuous problems.

   Milestone: An agent that can run on continuous action space. Evaluation results in the form of diagrams and tables.
3. **Week 5-6 (03/01/22-16/01/22)**:
   - Buffer period to wrap up.

Lent Term:

1. **Week 1-2 (17/01/22-30/01/22)**:
   - Prepare progress report and presentation.

   Milestone: Progress report.
2. **Week 3-4 (31/01/22-13/02/22)**:
   - Extension 1: evaluate the agent on MuJoCo

   Milestone: Run the agents on a chosen environment of MuJoCo and obtain evaluation results.
3. **Week 5-6 (14/02/22-27/02/22)**:
   - Extension 2: the number of atomic actions

   Milestone: Evaluation results in the form of diagrams and tables.
4. **Week 7-8 (28/02/22-13/03/22)**:
   - Start writing dissertation: Introduction + Preparation Parts write-up.
   - Extension 3: policy to select states to expand.

   Milestone: Introduction and Preparation Parts sent for review.

Easter Holiday:

1. **Week 1-2 (14/03/22-27/03/22)**:
   - Continue on extensions
   - Implementation Part write-up

   Milestone: Implementation Part sent for review. Wrap up on the extensions.
2. **Week 3-4 (28/03/22-10/04/22)**:
   - Continue on extensions

- Evaluation + Conclusion Parts write-up

Milestone: Full draft of dissertation. Send to supervisors and DoS for review.

3. **Week 5-6 (11/04/22-24/04/22)**:
   - Review dissertation and get feedback

   Milestone: Final draft of dissertation for proof-reading.

Easter Term:

1. **Week 1-2 (25/04/22-08/05/22)**:
   - Edit dissertation according to feedback

   Milestone: Final dissertation for submission.

Resources required

I will be using my personal laptop (MacBook Pro 2019, 1.4 GHz Quad-Core Intel Core i5, 8 GB, Intel Iris Plus Graphics 645 1536 MB with macOS Catalina) as my main working device. I will use Github for backup of code and version control. I will be using Overleaf and Google Docs for the writing of my dissertation, and backup with Google Drive. I will use public libraries such as PyTorch to support the development of my code. My personal laptop will be sufficient to run CartPole, Acrobot, and MountainCar environments, but when it comes to MuJoCo, I will use GPUs from the bioinformatics groups at the Computer Laboratory if my laptop is not sufficient.