BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EVOLUTIONARY ALGORITHMS IN REINFORCEMENT LEARNING

EVOLUČNÍ ALGORITMY V POSILOVANÉM UČENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. SABÍNA GULČÍKOVÁ**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

165202

Institut:	Department of Computer Systems (DCSY)
Student:	**Gulčíková Sabína, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Evolutionary Algorithms in Reinforcement Learning**
Category:	Artificial Intelligence
Academic year:	2024/25

Assignment:

1. Familiarize yourself with reinforcement learning, evolutionary algorithms (EA), and using EAs in reinforcement learning applications.
2. Propose a method integrating an EA into the selected application(s) of reinforcement learning to improve the results of reinforcement learning, for example, in terms of quality and/or interpretability.
3. Implement the proposed method as an extension of existing EA and reinforcement learning tools.
4. Evaluate the method on selected case studies. Compare the obtained results with the implementations in which no EA is used to improve reinforcement learning.
5. Discuss the results obtained in the project.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Items 1 and 2 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Sekanina Lukáš, prof. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	31.10.2024

Abstract

Reward function is one of the key signals guiding agents during learning in a model-free reinforcement learning setup. Translating a complex task objective into a meaningful mathematical expression is a non-trivial process, which, if done incorrectly, can lead to negative side effects such as reward hacking or complete failure to learn. Reward shaping is a method of providing additional information about the task to improve learning efficiency and stability, offering potential for alleviating problems stemming from incorrect reward design. This thesis explores the use of genetic programming for evolving reward shaping functions, shifting the design burden from manual engineering to the evolution of reward functions guided by a fitness function. The fitness function allows for direct optimization of desired agent behaviors and smooth learning dynamics, letting evolution discover suitable reward transformations. We evaluate this approach on the CartPole control task, and compare it against randomly obtained and manually designed shaping reward functions, as well as shaping-free approaches. In addition to standard learning, we examine the application of evolved functions in a transfer learning scenario, evaluating their robustness and impact on the agent's ability to learn in an environment with modified dynamics without the need for further hyperparameter tuning.

Abstrakt

Funkcia odmeny je jedným z najdôležitejších signálov, ktoré vedú agenta pri učení v konfigurácii bez znalosti modelu prostredia. Prenesenie komplexného cieľa úlohy do ekvivalentného matematického výrazu je netriviálny proces, ktorý pri nesprávnom prístupe môže viesť k neželaným javom, ako je zneužívanie odmeny (reward hacking) alebo úplné zlyhanie učenia. Formovanie odmien (reward shaping) je metóda, ktorá agentovi poskytuje dodatočné informácie o úlohe s cieľom zlepšiť efektivitu a stabilitu jeho učenia. Táto diplomová práca sa zaoberá použitím genetického programovania na evolúciu formovacích funkcií odmeny, čím presúva záťaž návrhu z manuálneho procesu na automatickú evolúciu riadenú vhodne navrhnutou fitness funkciou. Tá umožňuje optimalizáciu zameriavať na požadované správanie agenta a priebeh učenia, výsledkom čoho je možnosť objaviť vhodné formy odmeňovania. Kvalitu tohto prístupu vyhodnocujeme na úlohe CartPole a porovnávame ho s prístupmi založenými na náhodne vygenerovaných aj ručne navrhnutých funkciách, ako aj so základným prístupom bez dodatočnej funkcie odmeny. Okrem štandardného učenia vyhodnocujeme aplikácie vyvinutých funkcií v rámci tzv. transfer učenia, pričom sa zameriavame na ich robustnosť a dopad na agentovu schopnosť učiť sa aj v prostrediach so zmenenou dynamikou, bez potreby dodatočnej zmeny hyperparametrov.

Keywords

reinforcement learning, evolutionary computation, reward shaping, evolutionary reward design, DQN algorithm, PPO algorithm, CartPole task, generalization

Kľúčové slová

posilované učenie, evolučné výpočty, evolučné posilované učenie, evolučný dizajn hodnotiaceho signálu, DQN algoritmus, PPO algoritmus, CartPole úloha, generalizácia

Reference

GULČÍKOVÁ, Sabína. *Evolutionary Algorithms in Reinforcement Learning.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Rozšírený abstrakt

Posilované učenie (reinforcement learning, RL) je výpočetný prístup ktorý umožňuje agentovi naučiť sa optimálne správanie pomocou interakcie s prostredím na základe spätnej väzby vo forme odmeňovacích signálov. Na rozdiel od prístupu učenia s učiteľom, agent nepracuje priamo s anotovanými dátami, ale efektívne stratégie musí objavovať na základe skúseností a spätnej väzby z prostredia. Spätná väzba vo forme odmeňovacích signálov je však často oneskorená, alebo veľmi sporadická (v tomto kontexte tzv. *riedka*). Návrh vhodného signálu odmeny je teda kľúčový, no zároveň náročný proces, ktorý výrazne ovplyvňuje kvalitu a stabilitu učenia. Nesprávne navrhnuté odmeny môžu viesť k nežiadúcim javom ako je *reward hacking*, pomalé učenie, či tzv. katastrofické zabúdanie. *Reward shaping*, alebo formovanie odmien je technika, ktorá modifikuje pôvodný odmeňovací signál pridaním dodatočných, priebežných odmien, čím poskytuje agentovi častejšiu a informatívnejšiu spätnú väzbu. Cieľom je zlepšiť efektivitu agentovho učenia a stabilitu jeho správania bez nutnosti manuálneho ladenia odmien, ktoré často vyžaduje hlboké doménové znalosti.

V tejto práci sa zameriavame na automatizáciu návrhu odmeňovacích funkcií pomocou genetického programovania, ktoré prostredníctvom simulovania procesu evolúcie vyvíja matematické výrazy pre túto aplikáciu. Na testovanie prístupu sú použité dva RL algoritmy: Proximal Policy Optimization (PPO) a Deep Q-Network (DQN). Experimenty sú vykonané v prostredí CartPole s riedkou odmenou, predstavujúc úlohu s minimálnou informačnou hodnotou základného signálu odmeny. Funkcie získané evolúciou sú porovnávané s manuálne navrhnutými a náhodne vygenerovanými funkciami, rovnako ako s nastavením bez akejkoľvek pridanej informácie. V ďalších experimentoch sa sústreďujeme na použiteľnosť vyvinutej funkcie v prostrediach so zmenenou dynamikou.

Výsledky experimentov ukazujú, že evolučný prístup výrazne zrýchľuje učenie a zlepšuje jeho stabilitu. V niektorých prípadoch agenti s evolučne navrhnutou funkciou odmeny konvergujú až 15-násobne rýchlejšie než agenti v základnom nastavení. Evolučne navrhnuté funkcie tiež efektívne odstraňujú problémy ako katastrofické zabúdanie, a zmierňujú časté výkyvy vo výkone, ktoré sa objavujú pri použití náhodne vygenerovaných funkcií. Zatiaľ čo náhodne získané reprezentácie prinášajú určité zlepšenie voči základnému nastaveniu, evolúcia je schopná dosiahnuť lepšie výsledky najmä v on-policy (PPO) nastaveniach, čo naznačuje potrebu ďalšieho testovania v zložitejších prostrediach na zvýraznenie rozdielov medzi týmito dvoma prístupmi. Významnou výhodou evolučného prístupu je však úspešná generalizácia vyvinutej funkcie na modifikované verzie prostredia bez potreby dodatočného ladenia agenta. Tento fakt umožňuje znížiť záťaž spojenú s dotrénovaním a ladením parametrov na prispôsobenie sa charakteru prostredia, a s potrebou hlbokých doménových znalostí a schopnosti efektívne a dostatočne presne reprezentovať cieľ úlohy matematickým výrazom.

Získané výsledky povtrdzujú potenciál navrhnutej metódy najmä v on-policy prístupoch, a nechávajú priestor pre ďalší výskum. Ten by mohol rozšíriť aplikáciu metódy do komplexnejších prostredí, ako *MuJoCo* [1], a skúmať scenáre pôsobenia agenta v rôznych úlohách, či v kontexte kontinuálneho učenia. Evolúcia viacerých shaping funkcií súčasne, alebo využitie obmedzenia na potenciálovo založený shaping môže priniesť ďalšie zlepšenia v robustnosti a výkonnosti. Automatizácia návrhu odmien má potenciál výrazne znížiť riziko reward hackingu a nežiaduceho správania agentov, čo je kľúčové pre bezpečné nasadenie RL systémov v reálnych aplikáciách ako sú autonómne vozidlá, robotika, či rozhodovacie systémy založené na umelej inteligencii. Táto práca tak prispieva k rozvoju efektívnejších, stabilnejších a ľahšie prispôsobiteľných RL agentov.

Evolutionary Algorithms in Reinforcement Learning

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Lukáš Sekanina, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. I further acknowledge that available AI-based tools were used for minor language refinement and grammar correction. The intellectual content and ideas presented in this thesis are entirely my own work, and no AI tools were used to generate or develop the conceptual content and results of experiments.

.....
Sabína Gulčíková
May 18, 2025

Acknowledgements

I would like to express gratitude to my thesis supervisor, prof. Ing. Lukáš Sekanina, Ph.D., for his guidance, support, and valuable feedback throughout the course of this work. I am also deeply thankful to Tibor K. for his proofreading, relevant suggestions, and constant encouragement, and to my family for their unwavering support, patience, and for giving me all the space I needed to focus on my work.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Contents

Chapter 1

Introduction

Reinforcement learning (RL) is a computational framework concerned with learning optimal behavior through trial and error in an interactive *environment*. Unlike supervised learning, RL operates without a teacher, oracle, labeled or even predefined training data. The agent must actively interact with its environment to gather information and discover effective strategies, despite not knowing the goal beforehand. Instead, it must infer it from the sparse reward signals provided during training. This black-box environment communicates with the agent through scalar rewards, which are part of the environment's feedback as a response to the agent's actions. Such an open-ended setup introduces significant uncertainty and enormous room for error. However, the promise of RL to automate complex and tedious tasks makes it vital to address the challenges inherent to this learning paradigm. These challenges include poor initialization, unstable learning, and undesirable effects such as catastrophic forgetting, which is often linked to non-stationary environments, poor generalization, and the lack of long-term memory in RL systems.

Inspired by behavioral psychology, RL relies on feedback in the form of rewards to guide the *agent's* learning. Early behavioral experiments with animals demonstrated how structured reinforcement, such as receiving food when a lever is pressed, could effectively shape their behavior. This principle, called shaping, reinforces intermediate steps toward a final goal and remains an essential concept in both biological and artificial systems. In RL, shaping is typically employed in the domain of reward design and is referred to as *reward shaping* – the practice of modifying the reward function to provide more structured feedback to the agent. Since the reward is the main feedback signal through which an agent interprets success or failure, it provides a natural interface for injecting prior knowledge. However, the reward design is a nontrivial process. Poorly designed rewards can lead to unintended behaviors, such as *reward hacking*, where the agent maximizes the misleading or misaligned signal. Therefore, crafting effective reward functions is often a delicate and domain-specific task.

This thesis investigates an alternative: **automating the design of reward shaping functions using genetic programming** (GP). Instead of manually encoding knowledge into the reward signal, we use GP to evolve mathematical expressions that serve as reward functions. The key idea is that while the task-specific reward design is difficult, it may be easier to define a fitness function that evaluates desirable training characteristics, such as smooth convergence, high final performance and behavioral stability. By optimizing for such criteria, we aim to evolve reward functions that improve learning dynamics without compromising the ability to generalize. We focus particularly on whether evolved shaping functions can mitigate catastrophic forgetting and improve convergence. We also explore

whether these functions can generalize across environments with altered dynamics – such as changes in physical parameters – without re-tuning the agent. This tests whether shaping functions can encode higher-level behavioral priors, allowing agents to adapt to new settings with the same reward structure.

To test these hypotheses, we use two well-known RL algorithms: Proximal Policy Optimization and Deep Q-Networks, and evaluate their performance on the sparse-reward control task CartPole. Sparse rewards provide a minimal baseline, making any additional shaping signal more informative and impactful. We compare GP-evolved shaping functions to randomly generated shaping functions and the setup with no shaping, asses their impact on different agent types (on-policy and off-policy), and test generalization by transferring the shaping function across the environments with modified dynamics. The contributions are as follows: we propose an automated approach for evolving static extrinsic reward shaping functions using genetic programming; define a fitness function that prioritizes smooth convergence, high final performance, and reduced catastrophic forgetting; we evaluate the proposed approach across two reinforcement learning algorithms and confirm its ability to improve learning behavior; and explore the generalization of evolved shaping functions to environments with modified dynamics.

The rest of the thesis is structured as follows. Chapter 2 introduces the reinforcement learning framework, highlighting its key principles and differences from supervised learning. Chapter 3 presents evolutionary computation and its applications in RL. Chapter 4 introduces reward engineering and reward shaping, and briefly reviews existing techniques and limitations. Chapter 5 outlines the proposed method, including the genetic programming setup, metrics, and baselines. Chapter 6 presents the experimental findings and discusses the strengths and limitations of the proposed method. Finally, we conclude with a discussion of future directions for reward evolution in RL.

Chapter 2

Reinforcement Learning

In recent years, the field of artificial intelligence has witnessed many groundbreaking advances, one of them being the event of AlphaGo [89] defeating the world's best human player in the complex board game of Go. This achievement was soon surpassed by AlphaZero [90], an improved version of AlphaGo that mastered Go, chess, and shogi without any supervised learning on human knowledge, defeating AlphaGo by an impressive margin of 100-0 in direct matches. Shortly thereafter, OpenAI's Dota 2 bot demonstrated similar ability, defeating professional players in a highly complex real-time strategy game [73].

The mechanism that lays the basis for all these systems is the *reinforcement learning* (RL) framework. This enables agents to learn optimal behaviors through interactions with their environment. RL's formal framework defines these interactions in terms of states, actions, observations, and rewards, which serve as abstract representations of cause-and-effect relationships, uncertainty, non-determinism, and the pursuit of explicit goals. Reinforcement learning is particularly compelling because it directly reflects how humans and animals learn from their surroundings. By trial and error, together with feedback in the form of rewards or penalties, agents gradually refine their strategies to maximize long-term benefits. This makes such an approach well suited to solving problems that require sequential decision making in dynamic and uncertain environments.

In simple words, reinforcement learning can be understood as the science of decision making. Its main characteristics include the lack of a supervisor, since there is no oraculum to tell the agent which decision is right in the given state and time frame. This fact is what distinguishes reinforcement learning from supervised one, as only partial feedback is given to the learner about its predictions. Moreover, supervised learning typically assumes that training data are *independent and identically distributed (i.i.d.)*, which means that each data point is drawn independently from the same distribution and does not depend on previous inputs or predictions. In contrast, reinforcement learning violates this condition. The data from which the agent learns depend on its own actions and evolve over time. Agent's decisions must be made based on trial and error, and value of the reward signal. Further, feedback is often delayed, and not instantaneous. In many situations, it is not optimal to act greedily, as a momentary good decision might need to be given up for future ones with bigger impact. Time is also an important factor in this setting, as it represents a sequential decision process, taking place within a dynamic system. The actions taken by the agent influence the environment, which impacts the data it will see and receive in return.

In this chapter, we introduce fundamental principles in reinforcement learning and give the formal definitions of its key components. In addition, we give a brief look into classic

approaches to solving reinforcement learning problems. Similarly to other fields of artificial intellgence, there is a tension between breadth of applicability and mathematical tractability. We give formal definitions for the concepts for better understanding of the problem.

2.1 Preliminaries

Reinforcement learning is a broad field of study in machine learning, where agents learn to make decisions by interacting with an environment to maximize cumulative rewards. *Reinforcement learning problem* is a particular formulation or instance of a decision-making challenge being solved by RL. It is a straightforward framing of the problem of learning from interaction to achieve a goal. The two terms are often used interchangeably. The entirety of a reinforcement learning problem can be described by a simple agent-environment loop summarized in Figure 2.1. An agent receives the controlled environment's state and a reward associated with the last state transition. It then calculates an action which is sent back to the system. In response, the system makes a transition to a new state and the cycle is repeated. The learning problems differ in the particular settings of how the data are collected and how performance is measured. In recent years, *deep reinforcement learning (Deep RL)* has emerged as a powerful subclass of RL, where deep neural networks are used as function approximators. They are typically applied for representing policies, value functions, or environment models. This enables agents to handle high-dimensional input spaces (such as raw pixels), and complex tasks, but also introduces challenges related to stability, sample efficiency, and catastrophic forgetting. These issues are further discussed in Section 2.3.

In the following, we provide an explanation for the individual ingredients of this framework. The majority of notation and definitions were adapted from Barto and Sutton's textbook on reinforcement learning [96], Szepesvari's book on algorithms in reinforcement learning [97] and notes from David Silver's class on reinforcement learning [88]. Note that in other engineering fields, the terms *agent*, *environment* and *action* are often substituted by terms *controller*, *controlled system*, and *control signal*. Since reinforcement learning is an umbrella term for a vast set of problems, with an abundance of algorithms, hierarchies, and various settings for each of its components, we focus only on those relevant to our proposed method. For a deeper dive into the theoretical foundations and comprehensive overview of the tools and techniques on RL methods, together with their characterization in different optimization contexts, see [13, 14, 16, 36, 74, 98].

Agent and Environment. An autonomous *agent* refers to an entity making independent choices about acting in its environment without influence from a supervisor, or a general plan. In an applied context, an agent is an implementation of particular RL algorithm such as PPO, or DQN. This algorithm describes the approach to interaction with environment, computation, and further treatment of elements such as policy, or value function, and general structure of used computational structures and their interaction.

The *environment* can be thought of as a set of states S, in which the agent can occur. State $s \in S$, is a summary of information used to determine what happens next. It comprises sets of possible observations resulting from feedback signals given by the environment's response to the agent's behavior. There is a continual interaction of agent and the environment, the agent selecting actions with the environment responding to them and presenting its new settings. The complete specification of the environment defines *task*, an

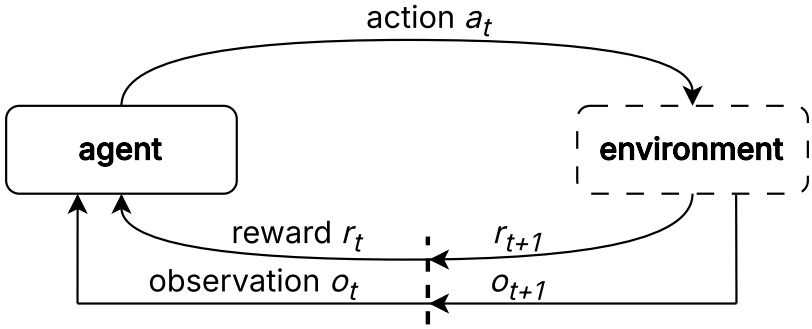


Figure 2.1: **Agent-environment loop.** At each time step t, the agent receives some representation of the environment's *state* s_t, and on that basis selects an *action* $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available in state s_t. In the subsequent time step, as a consequence of its action, the agent receives a numerical *reward* r_{t+1}, and finds itself in a new state, s_{t+1}.

instance of the reinforcement learning problem. It is important to distinguish between the terms *observation* and *state*. An observation refers to partial or incomplete information available to the agent, typically due to the inherent limitations or design of the environment or task. In contrast, a state represents the complete underlying information about the environment, which may not be directly observable by the agent.

Markov Decision Process. In formal terms, an environment of an RL problem can be framed as a Markov Decision Process (MDP), in which all states are *Markov*. A state s_t is *Markov* if and only if

$$\mathbb{P}[s_{t+1} \mid s_t] = \mathbb{P}[s_{t+1} \mid s_1, ..., s_t],$$

where $\mathbb{P}[s_{t+1} \mid s_t]$ represents the *probability* of state s_{t+1} given state s_t. *Markov property* states that *the future is independent of the past given the future*, i.e., each state with Markov property depends only on its direct predecessor. MDP $\mathcal{M}$ is then defined as a 5-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ denotes a finite set of states,
- $\mathcal{A}$ denotes a finite set of actions. An action $a \in \mathcal{A}$ represents a possible decision the agent can take, and a_t denotes the action taken at time step t,
- $\mathcal{P}$ is a transition probability matrix, $\mathcal{P}^a_{ss'} = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a]$, for state s and its successor s', such that for a fixed state s, $\sum_{q \in \mathcal{S}} \mathcal{P}^a_{sq} = 1$,
- $\mathcal{R}$ is a reward function, $\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$,
- γ is a discount factor $\gamma \in [0, 1]$.

The *discount factor* allows for specifying the present value of the future rewards. Intuitively, if $\gamma = 0$, immediate reward is preferred, while $\gamma = 1$, represents maximally far-sighted behavior, in which the agent cares about all rewards collected infinitely far in the future. In such case, it is assumed that the MDP contains an absorbing terminal state. This factor introduces the possibility of judgement whether short-term or long-term award

is preferred, and by what ratio. The closer to zero the value is, the more is immediate reward preferred. When using MDP as formalism, the environment is *fully observable*, meaning that the current state completely characterizes the process, and it is also a sufficient statistics of the future.

Model. In RL, the model is the descriptor of the environment from which we can learn or infer how the environment behaves in interaction with the agent in terms of providing the feedback. It encompasses two key functions: *the transition probability function* $\mathcal{P}$, and *the reward function* $\mathcal{R}$.

- **Transition function.** Transition function is a component of RL essential in the context of Markov Decision Process. Formally, it defines the dynamics of the environment by specifying the probabilities of transitioning between states, thus describing how the environment evolves as a result of the agent's actions. This function can be *deterministic*, where each action leads to a predictable state, or *stochastic*, where the actions lead to states probabilistically.

- **Reward function.** The reward function is a fundamental component of RL, which defines how an agent receives feedback from the environment based on its action. It assigns a numerical value to each state-action pair, indicating how desirable a particular outcome is. Given the properties of the function, it can be classified as *sparse*, or *dense*. A sparse reward function provides feedback infrequently, making it more difficult for the agent to learn, while dense provides a frequent feedback. The motivation behind the sparse reward function is to simplify the feedback signal, emphasize long-term objectives, and avoid the possible introduction of learning biases. However, sparse rewards pose one of the biggest challenges in reinforcement learning. Multiple methods have been proposed to enable learning with them [2, 100]. In this thesis, *reward* refers to the numerical feedback received from the *reward function (signal)*. In the domain of psychology, where the idea for the reinforcement of artificially intelligent agents comes from, rewards are said to be *"objects or events that make the subject return for more"* [87, 86], whereas reward signals are produced by reward neurons in the brain. To give an example, in a simple grid-world task, the agent must navigate a grid to reach a goal state. The reward function might give following reward:

$$\mathcal{R}^a_{ss'} = \begin{cases} +10 & \text{if } s' \text{ is the goal state,} \\ -1 & \text{otherwise.} \end{cases}$$

Based on these characteristics of the environment model, reinforcement learning algorithms can be broadly classified into *model-based*, and *model-free* reinforcement learning. While model-based methods explicitly leverage the reward and transition functions by planning, model-free approaches focus solely on learning through interaction, bypassing the need for an explicit model. In this context, the reward function still serves as the primary signal that guides the agent's exploration and exploitation, even in the absence of a model.

Model-based RL. In this approach, the agent maintains a direct model of how the world works in its representation, which can be established by using explicit transition and reward functions. This enables the agent to plan its action by simulating the outcomes of potential decisions, it is equivalent to planning with perfect information. In this case,

the model is either known or the algorithm learns it explicitly. In an example, planning algorithms such as *value iteration* [9], or *Monte Carlo Tree Search* [17] rely on a detailed model of the environment to predict future states and rewards. When the environment is fully known, the optimal solutions can be found using *dynamic programming* [10]. Model-based RL often excels in scenarios where a good approximation of the environment can be obtained, enabling efficient planning and decision making. However, constructing and maintaining such a model can be computationally expensive and prone to inaccuracies in environments with high stochasticity or partial observability.

Model-free RL. In contrast, this approach avoids the need for an explicit model. Instead, the agent directly learns the policy or value function from the interactions with the environment. Methods such as *Q-learning* or *policy gradient algorithms* focus on optimizing the agent's behavior by trial and error. While these are often simpler to implement, they require significantly more data to learn effectively, especially in environments with sparse rewards or complex dynamics. Although model-free methods are widely applicable, their reliance on trial-and-error learning can lead to high sample inefficiency. Techniques like *Deep Q-Learning* [66] and *Proximal Policy Optimization* (PPO) [84] address these issues by combining deep learning with improved exploration strategies.

Key Functions. The agent environment loop gives rise to multiple functions that the agent tries to optimize, serving as signals carrying information about the nature of the entire problem. These functions, namely *policy* and *value function* encapsulate the agent's objectives, strategies and expectations within the environment. Together, they form the core of what is learned and refined throughout the reinforcement learning process.

- **Policy.** Policy π is a mapping from state $s \in \mathcal{S}$ to action $a \in \mathcal{A}(s)$, which can be either *deterministic*, $\pi(s) = a$, or *stochastic* $\pi(a \mid s) = \mathbb{P}_\pi[A = a \mid S = s]$. This function is used to describe the behavior of the agent, defining how the agent should act in any given situation, and is often thought of as the brain of the agent. In practice, the policy is usually a neural network that takes the current state of the game as input and calculates the probability of taking any of the allowed actions. An actual policy function might have millions of parameters, so the task comes down to finding the precise setting of these parameters such that the policy plays well (i.e., wins a lot of games). Furthermore, based on how the agent interacts with its policy, there are two main categories of learning, namely, *on-policy* and *off-policy learning*.
  - **On-policy learning.** In this approach, the agent updates its policy based on the actions it has taken. In this case, the policy used to select actions and the policy that is being improved are one and the same. Algorithms such as *SARSA* (State-Action-Reward-State-Action) and *Policy Gradient methods* fall into this category. Since the agent's experience is directly influenced by the policy being evaluated, on-policy methods typically require more interactions with the environment to converge to an optimal policy.
  - **Off-policy learning.** In off-policy learning, the agent can update its policy based on actions taken by a different policy. This means that the agent can learn from actions taken by another agent or a past version of itself. *Q-learning* is a common off-policy algorithm. These methods allow the agent to explore the environment more efficiently and reuse past experiences, making them more

data-efficient. However, they can be less stable because the learning process may not always reflect the true consequences of actions taken under the current policy.

- **Value function.** The value function $V^\pi(s)$ represents the expected cumulative reward the agent can achieve starting from state s, following a policy π. This function summarizes the *desirability* of states in terms of expected future rewards. It gives the long-term value of state s. Informally, the *value* of a state s, under the policy π denoted $V^\pi(s)$ is the expected return when starting in s and following π thereafter. Formally, we define $V^\pi(s)$ as:

$$V^\pi(s) = E_\pi\{\mathcal{R}_t \mid s_t = s\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\},$$

  where E_π denotes the expected value given that the agent follows policy π. The notion of *state-value function* can be extended to *action-value function* $Q^\pi(s, a)$ defined as the expected return starting from state s, taking action a, and following policy π:

$$Q^\pi(s, a) = E_\pi\{\mathcal{R}_t \mid s_t = s, a_t = a\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\}.$$

- **Optimal value function and policy.** Solving a reinforcement learning task comes down to finding a policy that achieves optimal reward during the duration of the task. In the context of MDPs, this model is solved once the optimal value function is identified. This function specifies the best possible performance in such a model. Value functions define a partial ordering over policies, where $\pi \geq \pi'$ if and only if $V^\pi(s) \geq V^{\pi'}(s)$, for all $s \in \mathcal{S}$. The *optimal state-value function* $v^*(s)$ is the maximum value function over all policies defined as:

$$v^*(s) = \max_\pi v_\pi(s).$$

  The *optimal action-value function* $q^*(s, a)$ is the maximum action-value function over all policies defined as:

$$q^*(s, a) = \max_\pi q_\pi(s, a).$$

  An optimal policy $\pi^*(a \mid s)$ can be found by maximizing over $q^*(s, a)$, such that:

$$\pi^*(s \mid a) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q^*(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

Types of Environments. The type of environment plays an important role in determining the complexity of the agent's task and the choice of algorithms used for its solution. Environments can vary in terms of how much information they reveal to the agent (so-called observability) and the nature of their dynamic (stochasticity). Understanding these differences is essential for designing effective reinforcement learning systems.

Observability. The aforementioned MDP formalism is characterized as a fully observed environment in which the agent is able to directly observe the state of the environment. When working with partially observable environments, the agent observes the environment indirectly and works with belief states, instead of having direct access to the true states. A practical example of this is a robot with camera vision, not told about its absolute location, or a trading agent observing solely current stock prices. In such an application, the agent's internal representation of state differs from the actual environment's state. In formal terms, such a model can be described by a *partially observable Markov decision process* (POMDP) [3]. This model is an MDP with hidden states.

Stochasticity. Here we classify the environments based on their dynamics. In a *deterministic environment*, the outcome of an agent's action is entirely predictable. Given a state s and action a, the successor state s' is uniquely determined by the transition function $T(s, a)$. In a *stochastic environment*, the outcome of an agent's action is probabilistic. The next state depends on the transition probabilities $\mathbb{P}(s' \mid s, a)$.

Types of Agents. Based on how the value function and the policy are represented and utilized, reinforcement learning agents can be categorized into the following categories. For a better understanding of how individual approaches and algorithms relate to each other, refer to Figure 2.2.

Value based. A value-based agent focuses primarily on learning a value function $V^\pi(s)$, or $Q^\pi(s, a)$, to estimate the expected cumulative reward for states or pairs of actions of states. The policy is derived implicitly by greedily choosing actions that maximize the value function. Such an approach is efficient in environments with well-defined reward signals that are easily propagated through states. An example of an algorithm employing such approach is *Q-learning* [101].

Policy based. A policy-based agent explicitly represents and learns $\pi(a \mid s)$ policy without directly maintaining the value function. This policy is often parameterized by a neural network. This approach is better suited for continuous action spaces and can model stochastic policies, which are critical in environments requiring diverse strategies. An example of algorithms based on this principle is *REINFORCE* [102].

Actor critic. The actor critic approach combines the strengths of both aforementioned approaches. This agent maintains an *actor*, which represents the policy $\pi(a \mid s)$, and a *critic*, which estimates the value function (either $V^\pi(s)$ or $Q^\pi(s, a)$). The critic evaluates the actor's actions by providing feedback in terms of value estimates that are then used by the actor to improve the policy. This approach allows for stable and efficient learning in both continuous and discrete spaces; however, it is more complex to implement and tune, with additional computational overhead due to maintaining both components. An example of algorithms based on this principle is *Proximal Policy Optimization* [84], *Advantage Actor-Critic* (A2C) [62, 4], or *Deep Deterministic Policy Gradient* (DDPG) [8].

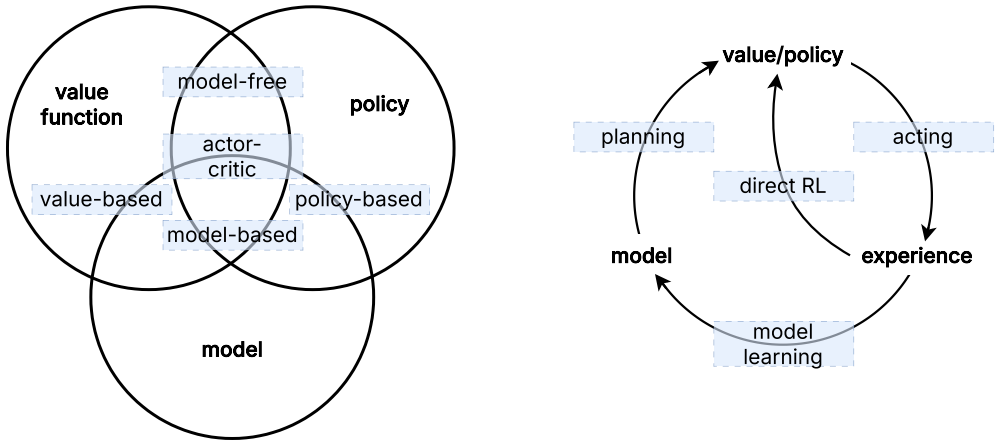


Figure 2.2: **Taxonomy of RL approaches.** The left diagram represents a taxonomy of reinforcement learning (RL) approaches based on value function, policy, and model. These components form overlapping categories, indicating how different RL methods focus on modeling specific aspects of the decision-making process. The right diagram summarizes the iterative learning loop in RL, and depicts the relationships between the value/policy, model, and experience. In model-free RL, the agent directly uses experience to update its value function or policy, while in model-based RL, the agent uses experience to learn a model of the environment's dynamics. Once the model is learned, it can be used for planning, allowing the agent to improve its value function or policy without requiring direct interaction with the environment. Reproduced from [88].

2.2 Algorithms for Reinforcement Learning Problems

In this section, we introduce two widely used RL algorithms: Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). These methods are representative of two main RL paradigms, namely *value-based learning*, and *policy-based learning*, offering different approaches to learning from interaction with an environment.

DQN and PPO were selected for this thesis as they allow us to study undesirable phenomena such as catastrophic forgetting or poor generalization across different RL settings. As these algorithms differ in their training strategies and internal mechanisms, we briefly describe each to clarify where and how catastrophic forgetting may arise. Since this thesis focuses on improving learning efficiency and mitigating issues like catastrophic forgetting, we emphasize each algorithm's connection to the emegence of such problems.

2.2.1 Deep Q-Learning

Deep Q-Learning (DQN), often referred to as Deep Q-Network, was first used in a landmark paper that experimented with this approach applied to playing Atari games [63]. It is a *value-based*, *model-free* reinforcement learning algorithm built on optimal function for predicting the expected sum of rewards in an environment, known as Q-function. It combines Q-learning [20] with deep neural networks. The agent learns an *action-value* function $Q(s, a)$, which estimates the expected future reward of taking an action a in a given state s, and following the current policy thereafter. This algorithm works *off-policy*, since it learns

from actions generated by a behavior policy (typically ϵ-greedy[1]), and not necessarily the current policy.

Component Setup. The DQN architecture is based on several key components that interact during the agent's learning and acting process. In the following, we briefly describe the most important ones:

- **Experience replay buffer** — This component stores past transitions (s, a, r, s') encountered by the agent. Reusing these experiences during training improves sample efficiency and breaks the temporal correlations that would otherwise destabilize learning.
- **Q-network** — This is a deep neural network that approximates the action-value function $Q(s, a)$. Given an observation as input, it predicts the expected cumulative reward for each possible action. The agent uses this network during the *acting phase* to select actions.
- **Target network** — This component is a periodically updated copy of the Q-network. It is used during the *training phase* to compute target values, which improves learning stability by reducing fluctuations in the estimated targets.

Algorithm. The following description of the algorithm is adapted from [64], which introduces mechanisms such as the usage of two neural networks to improve the original version of the algorithm. The algorithm operates in two phases, *acting phase*, and *learning phase*.

- **Acting phase.** The agent interacts with environment and populates the replay buffer to gather experience. At each itmestep t:
  - The current state s_t is passed to the Q-network, which predicts the expected return $Q(s_t, a)$ for each possible action a.
  - An action is selected using an ϵ-greedy strategy to balance exploration and exploitation.
  - The environment returns the next state s_{t+1}, reward r_t and termination flag.
  - The tuple $(s_t, a_t, r_t, s_{t+1}, done)$ is stored in the experience replay buffer, where *done* is a boolean flag indicating whether s_{t+1} is a terminating state.
- **Learning phase.** Periodically, the agent updates its Q-network based on mini-batches sampled from the replay buffer:
  - A batch of transitions is sampled uniformly from the buffer.
  - For each transition, the target value y_t is calculated as:

$$y_t = \begin{cases} r_t + \gamma \max_{a'} Q_{\text{target}}(s_{t+1}, a') & \text{if the episode is not } done, \\ r_t & \text{otherwise.} \end{cases}$$

    where r_t is the reward received at time t, s_{t+1} is the next state, $\gamma \in [0, 1]$ is the discount factor, Q_{target} is the target Q-network, and a' is the action being maximized over.

[1] ϵ-greedy policy is an action selection strategy in which the agent chooses a random action with probability ϵ (emphasizes exploration), and the action with highest estimated value with probability $1-\epsilon$ (emphasizes exploitation).

– The Q-network is updated by minimizing the loss:

$$L = \mathbb{E}[(y_t - Q(s_t, a_t))^2],$$

using stochastic gradient descent, such that $Q(s_t, a_t)$ is the predicted Q-value for taking action a_t in state s_t.

– In every C steps, the weights of the target network are updated to match the current Q-network.

DQN & Catastrophic Forgetting. Since DQN uses an experience replay buffer, older experiences are often rewritten by new ones. Moreover, as updates are based on randomly sampled transitions, tasks not currently present in the buffer may fade from the memory, especially in sequential tasks or non-stationary setting. For example, this can lead to situations in which the agent may retain knowledge of successful behaviors while forgetting how to recover from rare or negative configurations.

2.2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a member of the proximal policy family of reinforcement learning algorithms [84]. Specifically, PPO belongs to the class of *policy gradient methods*, which directly optimize the policy function $\pi_\theta(a_t \mid s_t)$ to maximize expected future rewards (unlike value-based methods such as DQN). PPO is a policy-based (on-policy), model-free algorithm that uses stochastic gradient ascent to update the policy and improves training stability through a clipped surrogate objective that limits the size of policy updates.

It is important to note that unlike DQN, policy gradient methods are able to take continuous action spaces. This is because the actor can output a vector of means and variances parameterising a distribution over actions, and then sample actions from this distribution. On the other hand, the Q-network in DQN is only able to take in states and output a single scalar value for each discrete action.

Component Setup.

- **Actor-critic architecture.** PPO maintains two neural networks, *actor (policy network)* that outputs a probability distribution over actions and *critic (value network)*, which estimates the state value $V(s)$ for the calculation of advantages.
- **Advantage estimation.** PPO uses Generalized Advantage Estimation (GAE) [83] to compute an estimate $\hat{A}_t$, which measures how much better an action performed compared to the expected value of the state.
- **Clipped surrogate objective.** To avoid large policy updates, PPO maximizes a clipped objective that prevents the new policy from moving too far from the old one:

$$L^{\text{CLIP}}(\theta) = \frac{1}{|B|} \sum_t \min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right),$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the current and old policy. This objective ensures stable updates and prevents destructively large policy shifts. Detailed explanation of the objective can be found in the original paper [85].

- **Replay Memory.** PPO does not use a long-term experience replay buffer. Instead, it collects a fresh batch of experiences during each rollout and reuses this batch for multiple optimization epochs.

Algorithm. PPO alternates between a *rollout phase* and a *learning phase*. Unlike DQN, the buffer is replaced each iteration and reused several times for training, improving sample efficiency.

- **Rollout phase.** The agent interacts with the environment using the current (frozen) policy $\pi_{\theta_{\text{old}}}$ to collect a batch of transitions (s_t, a_t, r_t, s_{t+1}). These are stored temporarily for use in training.
- **Learning phase.** The agent performs several epochs of stochastic gradient ascent on minibatches drawn from the collected trajectories. The objective combines three loss components:
  - **Policy loss (clipped surrogate objective):** updates the actor network to improve policy while avoiding large deviations.
  - **Value function loss:** mean squared error between predicted value and empirical return, used to train the critic.
  - **Entropy bonus:** encourages exploration by maximizing the policy entropy.

  After training, the policy network parameters θ are copied to θ_{old}, and the cycle repeats.

PPO & Catastrophic Forgetting. Since PPO discards old experiences and trains solely on recent trajectories, it can be particularly vulnerable to forgetting in continual learning settings. The policy may rapidly adapt to new tasks, potentially at the cost of degrading performance on previously learned tasks.

2.3 Challenges in Reinforcement Learning

Due to the complexity involved in specifying and solving reinforcement learning problems, a number of inherent and practical challenges arise. These stem from the intricacy of real-world environments, the difficulty of designing effective reward functions, and the need for efficient learning in uncertain and dynamic settings. As these agentic systems grow in complexity, issues such as *exploration-exploitation trade-off*, *deadly triad*, or *sample inefficiency* need to be addressed to improve their performance and scalability. Additionally, practical concerns such as safety and generalization introduce additional layers of complexity that need consideration. This section offers a brief exploration of the aforementioned problems. Their deeper study can be found in [22, 96].

Exploration-Exploitation. The dilemma of exploration versus exploitation is one of the most discussed challenges when implementing reinforcement learning algorithms. *Exploration* is understood as any action that lets the agent discover new features about the environment, while *exploitation* is the act of capitalizing on the knowledge already gained. One has to think about the right balance between these two, as without enough exploration, the system cannot learn the environment well enough, while without exploitation, the reward optimization task cannot be completed.

Deadly Triad Issue. Reinforcement learning is a flexible framework where one must carefully choose the model and approach to address key components such as policy, value function, and transition dynamics. Although combining different methods can lead to powerful solutions, improper combination of complicated algorithms can introduce many problems. One such challenge is the deadly triad [99], a well-known issue in RL.

The deadly triad arises when TD-learning, off-policy learning and function approximation (e.g., using deep neural networks) are combined. While each of these approaches is powerful on their own, their combination leads to instabilities and even divergence during training. This is because off-policy learning relies on data collected under a different policy, which can exacerbate the approximation errors introduced by function approximators, especially when paired with bootstrapping's reliance on recursive value updates. To avoid this issue, several techniques, such as replay experience, have been introduced in modern algorithms such as *Deep Q-Networks* [41].

Sample Efficiency. Solving a reinforcement learning problem requires a massive number of interactions with the environment to learn optimal policy [103]. This poses a significant challenge in real-world applications, where collecting data can be costly and time consuming. For example, training a robotic arm to pick up objects may require thousands of trials. To address this, sample-efficient algorithms, such as model-based RL and methods that incorporate imitation learning [45], seek to reduce the number of required interactions by using predictive models or leveraging demonstrations from human experts.

Reward Hacking. Reward hacking, often referred to as specification gaming [92, 21], occurs when an agent learns to exploit flaws or ambiguities in the reward function to achieve high rewards, without genuinely learning or completing the intended task. In a practical example, an agent trained to walk in a simulated environment might learn to fall intentionally if falling provides a higher cumulative reward due to an oversight in the reward function. This phenomenon arises because the agent does not possess an intrinsic understanding of the task's broader context. Instead, it relies solely on its interaction with the environment and the evaluation provided by the reward function.

Learning and Generalization Challenges. Unlike supervised learning, reinforcement learning operates on non-i.i.d. data generated from sequential interactions with the environment. As a result, not only do classical challenges like overfitting emerge, but additional RL-specific issues—such as instability, forgetting, and poor generalization, also arise. In the following, we highlight key learning difficulties that this thesis seeks to mitigate through the proposed approach.

Overfitting and non-stationarity. Agents often overfit to specific training environments and fail to generalize to small variations [104]. In reinforcement learning, this is exacerbated by the fact that the training data distribution depends on the policy itself. As the policy changes, so does the data distribution, leading to instability and difficulty in transferring knowledge between tasks or even across time in a single task setting.

Transfer learning and generalization. Generalization is a quality of artificial system, which allows it to apply its knowledge to unseen data. In the setting of RL, this translates to an agent being able to accomplish a task different from that on which it had

Figure 2.3: **Graph illustrating within-task catastrophic forgetting in CartPole.** Episode length is a core indicator of task performance. After reaching consistently high episode lengths around 120,000 training steps, performance begins to deteriorate despite continued training. The graph shows periods of high reward followed by sharp drops, notably around 200,000 and 240,000 steps, indicating the onset of catastrophic forgetting. Values were smoothed using a Gaussian filter with $\sigma = 1$ for clarity.

been trained. This is particularly challenging in environments where agents deal with tasks that are either completely new or unnoticeably modified from the ones they were trained on. A practical example of this situation can be seen with an agent trained to navigate a maze. If the maze layout changes slightly, the agent must adapt its policy without starting from scratch. Algorithms such as meta-reinforcement learning [39] address this by training agents with the objective of learning new tasks more quickly.

Catastrophic forgetting. Catastrophic forgetting is typically studied in the context of task switching, or continual learning, where an agent loses performance on previously learned tasks when exposed to new ones [34, 50]. However, an equally problematic variant can occur even within a single task. In this scenario, agent trained on a fixed environment (e.g., CartPole introduced in Section 5.2.2) can show convergence followed by unexpected performance drops, despite no changes in the task configuration. This instability is often caused by internal factors such as value overestimation, unbalanced updates, or inference on learned policy representation [46]. In deep reinforcement learning, this issue is amplified by the use of function approximation, which overwrites previously learned knowledge when optimizing for short-term reward.

This thesis investigates this phenomenon in the CartPole environment. A representative training curve (see Figure 2.3) illustrates how standard PPO training achieves near-optimal performance early on, only to deteriorate again in later stages. This shows that achieving maximum reward temporarily does not guarantee that the agent has truly mastered the task. Since such performance degradation can stem from various factors and appears across different settings, we investigate whether a single shaping-based approach can consistently improve stability across algorithms, even under a fixed reinforcement learning hyperparameter setup.

Chapter 3

Evolutionary Computation

Evolutionary computation (EC) is a subset of artificial intelligence inspired by Charles Darwin's theory of natural selection. It translates the biological processes of variation, selection, and inheritance into a computational framework, forming the basis for a family of optimization strategies. These methods are called evolutionary because they mimic biological evolution, which can yield organisms with desired traits even without explicit learning during their lifetimes. Unlike reinforcement learning, evolutionary methods excel in scenarios where an agent cannot accurately sense or observe its environment, and they can be particularly effective when the problem's search space is manageable or well-structured.

Evolutionary algorithms (EAs) provide a flexible framework rather than a fixed solution, allowing customization to specific problems. Key elements include designing candidate solutions (individuals), formulating the fitness function, and selecting mechanisms to guide the evolutionary process. For example, strategies such as promoting elitism can preserve the best solutions across generations, while diversity-preserving methods help avoid premature convergence. Successful application of these algorithms requires deep domain knowledge to ensure that these components align with the problem's requirements and constraints. Due to their ability to optimize complex functions, evolutionary algorithms are applicable to a wide range of challenges, including mechanical design [53], electromagnetic optimization [79], environmental modeling [12], finance [54], and machine learning tasks such as hyperparameter tuning [82], neural architecture search [58], and automated feature selection [49].

In this chapter, we explore the structure and key components of evolutionary algorithms, with a focus on genetic programming, a specialized subset of EC particularly suited to the method proposed in this thesis.

3.1 Preliminaries

As stated before, the evolutionary computation is inspired by the Darwinian process of natural selection. This means that it follows several principles of Darwinian evolution that will be required as we implement these approaches in a computational setting [25]. In order for natural selection to occur as it does in nature, all three of these elements must be present. These include *heredity*, which means that there must be a process in place by which children receive the properties of their parents. If creatures live long enough to reproduce, then their traits are passed down to their children in the next generation of creatures. *Variation* is another important property of evolution, which ensures that variety

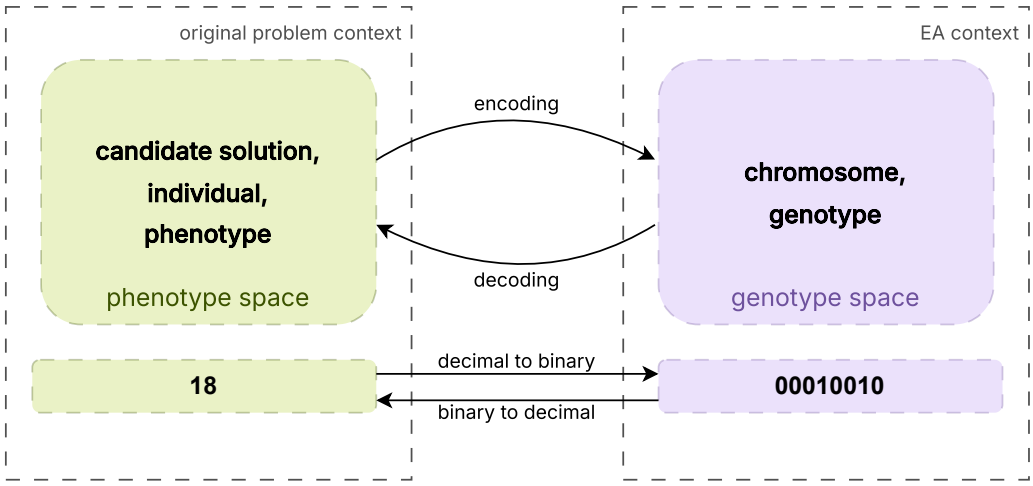


Figure 3.1: **Example transformation between problem representation and evolutionary algorithm operation**. An integer (18) from the phenotype space (original problem context) can be encoded as a binary string (00010010) in the genotype space (EA context) and decoded back.

of individuals is developed, and a new combination of traits, parameters and characteristics ensures the development and fact that children will not solely be exact copies of their parents. Lastly, the *selection* mechanism allows some members of a population to have the opportunity to be parents and transmit their genetic information. This is typically referred to as *the survival of the fittest*. Natural selection operates on the principle that some traits are better adapted to the environment of the creature and therefore produce a greater likelihood of surviving and reproducing. In the following, we discuss evolutionary algorithms in detail. In order to define a particular algorithm, one needs to specify a number of components, procedures, and operators. The design of each of these should be approached carefully, as it has a great impact on the efficiency, accuracy, and success of the algorithm.

Representation of Individuals. The first step in initializing an evolutionary algorithm is the proper link between the original problem context and the problem-solving space where evolution takes place. This involves simplifying or abstracting some aspects of the real world to create a well-defined problem context whose solutions can be evaluated. Objects forming possible solutions within the original problem context are called *phenotypes*, while their encodings, the individuals within the EA, are called *genotypes*. The problem of representation amounts to specifying a mapping from the phenotypes onto a set of genotypes. It is desirable for an inverse mapping from genotypes to phenotypes to be defined as well. It is necessary for the representation to be invertible so that for each genotype there is at most one corresponding phenotype. The solution – a good phenotype – is obtained by decoding the best genotype after termination. Example of the representation can be seen in Figure 3.1.

Population. Evolutionary algorithms are population-based, which means that they process a whole collection of candidate solutions simultaneously. The population in this context

is simply a multiset of different individuals or genotypes. Its role is to hold the representation of possible solutions, and it forms the unit of evolution. Although individuals are static objects that do not change or adapt, the population does. In most of the EA applications, the population size is constant and does not change during the evolutionary search. The *diversity* of a population is a measure of the number of different solutions present. This can be understood in terms of the number of different fitness values present, the number of different phenotypes, genotypes, or other statistical measures such as entropy. The population in a particular iteration of the algorithm is called generation.

Population Initialization. The initialization of population is usually done by seeding randomly generated individuals. However, given the additional computation cost, it is also possible to make use of problem-specific heuristics to create an initial population with higher fitness.

Fitness Function. The fitness function is a formalized representation of the task solved in an evolutionary context. It is a representation of the requirements that the ideal solution should meet. This is a crucial function, as it guides the whole process of evolution, forms the basis of selection, facilitates improvements, and defines their meaning. Practically, it is a function of assigning a certain value of quality to genotypes. In practice, this function often consists of an inverse representation to create the corresponding phenotype, followed by a quality measure in the phenotype space.

Parent Selection. The role of parent selection is to allow better individuals to recombine to create a new generation, and to choose individuals according to their quality. In evolutionary computation, parent selection is usually probabilistic, so high-quality individuals have a higher chance of becoming parents than lower-quality ones. The low quality needs to be given chance as well, since otherwise the algorithm might become greedy and get stuck in local optima. Several methods exist to select parents, including the *roulette wheel* and *tournament* selection mechanisms. In roulette wheel selection, individuals are assigned selection probabilities proportional to their fitness, similar to spinning a weighted roulette wheel. In contrast, tournament selection involves randomly selecting a subset of individuals and choosing the best one in this group to be the parent. Each method has its advantages and trade-offs, which can influence the algorithm's exploration-exploitation balance.

Variation Operators. Mutation and recombination (crossover) are variation operators whose role is to create new individuals from the old ones. This amounts to generating new candidate solutions to undergo evaluation and possibly become members of the next generation of genotypes. Based on their arity, variation operators are divided into unary (mutation), and n-ary (recombination).

Crossover. Crossover, or recombination, is a binary operation which merges information from two parent genotypes into one or two offsprings. Given the stochastic nature of this operator, the choice of what part of each parent are combined, depends on a random drawing. Recombination is usually applied with high likelihood. There exist variants with higher arity where more than two parents are combined. Despite the fact that these have no biological equivalent, some studies indicated their positive effect on evolution [23, 24]. Example approaches to recombination, namely *single-point crossover*, *two-point crossover*

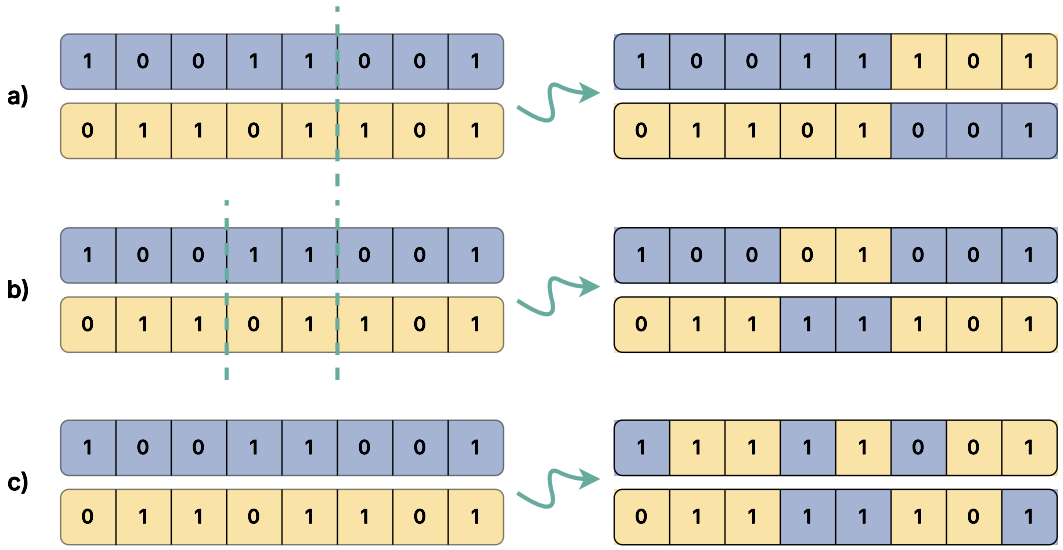


Figure 3.2: **Example of crossover of two genotypes.** a) *single-point crossover* with randomly chosen point of crossover, b) *two-point crossover* with two randomly generated points of crossover, and c) *uniform crossover*, where for each bit of the offspring, value from one parent is randomly chosen.

and *uniform crossover* are demonstrated in Figure 3.2. The presence and manner of this operation differ across different subdomains of evolutionary computation. It can be the only variation operator, the main search operator, or it can be omitted altogether. The principle behind this operator is that by mating two individuals with different desirable features, it is possible to produce an offspring that combines both qualities. It is important to note that variation is highly dependent on the manner of representation.

Mutation. Mutation is an operation applied to one genotype, which results in a slightly modified individual. This operator is always stochastic, its output – the offspring – depends on a series of random choices, as it is supposed to cause a random, unbiased change. In practice, when working with binary genotype encoding, after recombination of parents, a single bit of a phenotype might be flipped with a certain probability. This operation is supposed to introduce a fresh individual, promote the diversity of the population, and possibly jump to a new point in the search space. Mutation plays a different role in various branches of evolutionary computation. In some, it is used as the sole variation operator, while in others it is omitted completely.

Survivor Selection (Replacement). The survivor selection mechanism, although in principle similar to parent selection, occurs at a different stage of the evolutionary cycle – after the offspring have been created from the selected parents. Since population size is typically fixed, it is crucial to determine which individuals will carry over to the next generation. This decision is often based on fitness values. Unlike parent selection, survivor selection is generally deterministic. Two common strategies are *generational replacement*, where the new population consists entirely of offspring, replacing all individuals of the previous generation. The other approach, *steady-state replacement*, is a mechanism where

a fixed number of the best individuals from the previous generation are retained, while the remaining slots are filled with individuals from the offspring, ensuring the population size remains constant. Additional techniques, such as *elitism*, can be employed to further refine the algorithm. In elitism, the best individual from the previous generation is always preserved in the new population, ensuring that top-performing solutions are not lost during evolution.

Termination Condition. The termination condition of the algorithm can be approached in two ways. If the problem has a known optimal fitness level, equivalent to an optimum of the given objective function, in an ideal setting, the stopping condition would be the discovery of a solution with such fitness. However, EAs are stochastic, and many simplifications are often employed within the model of the real-world problem. This means that there might not be a guarantee of reaching an optimum. Therefore, this condition is often extended by options so that termination occurs after the elapsed maximally allowed CPU time, reaching a limit of the total number of fitness evaluations, population diversity dropping under a given threshold, or fitness improvement remaining stagnant, under a certain threshold, for a given period of time.

3.1.1 General Structure of Evolutionary Algorithm

The common underlying idea behind all techniques of evolutionary computation is as follows: given a quality function to be maximized, the process starts by randomly generating a set of candidate solutions, which represent elements of the function's domain. The quality function is then applied to these candidates, serving as an abstract measure of fitness, with higher values indicating better solutions. Based on their fitness, the most promising candidates are selected to form the basis of the next generation. This is achieved through recombination and/or mutation. Recombination involves combining two or more selected candidates (referred to as parents) to produce one or more new candidates (offspring). Mutation, on the other hand, operates on a single candidate, introducing small changes to create a new one. Together, these operations generate a set of new candidates, which are then evaluated for their fitness. The offspring compete with the existing population – often based on their fitness (and sometimes their age) – for a place in the next generation. This cycle of evaluation, selection, and variation is repeated iteratively until a solution meeting the desired criteria is found. The general structure of such algorithm can be seen in Algorithm 1.

Algorithm 1 General structure of evolutionary algorithm.

```
1: INITIALISE population with random candidate solutions;
2: EVALUATE each candidate;
3: repeat
4:     SELECT parents;
5:     RECOMBINE pairs of parents;
6:     MUTATE the resulting offspring;
7:     EVALUATE new candidates;
8:     SELECT individuals for the next generation;
9: until TERMINATION CONDITION is satisfied.
```

3.2 Genetic Programming

Genetic programming (GP) [51] is a specialized subfield of evolutionary computation, with a restricted form on representation of individuals. Although traditional evolutionary algorithms typically operate on fixed-length representations such as binary strings, vectors, or permutations, genetic programming evolves populations of tree-like structures. These structures can represent hierarchical entities such as symbolic mathematical expressions, computer programs, or logical rules. Individuals in GP are typically encoded as syntax trees, where nodes represent functions (e.g., mathematical operators such as $+$, $-$, or trigonometric functions) and leaves represent variables or constants. This hierarchical encoding naturally aligns with problems involving compositional structures, such as equations, rules, or programs. The selection schemes for chosing parents and survivors can be adapted from the classical evolutionary algorithms. Distinct approaches are adapted in the domain of crossover and mutation operators, which is necessary given the different mathematical structure used for representing the problem. List of possible approaches can be found in Table 3.1.

Unlike fixed-length representations, the tree structures in GP allow solutions to dynamically vary in size and complexity. This dynamic nature enables GP to explore a vast and flexible solution space, including representations that would be difficult or impossible to predefine manually. The solutions generated by GP are often interpretable, as they take the form of symbolic expressions or programs. This interpretability is a significant advantage in domains where understanding the behavior or rationale behind a solution is crucial, such as scientific discovery or the design of automated algorithms.

Evolving mathematical expressions. GP is well-suited for evolving symbolic mathematical expressions, making it an effective tool for generating reward functions in reinforcement learning. Its tree-based representations allows GP to search a vast space of diverse expressions that can influence agent behavior in nuanced way. For example, a simple shaping reward function might be evolved into expression such as

```
(div(add(sub(1, abs(div(x, 0.2095))), sub(1, abs(1))), 2).
```

The symbolic flexibility is a central feature of the method proposed in this thesis. Rather than seeking a single, optimal function, GP enables the discovery of multiple candidate reward functions, which can be evaluated in terms of their influence on learning dynamics and generalization. While these expressions are human-readable, they often take forms that a human designer might not intuitively create, ones featuring seemingly redundant terms, deep branches resulting in a constant value, or other mathematically obscure structures. Interestingly, recent developments in the theory of interpretability [67] suggest that such forms may not be accidental. Under the surface, artificially intelligent models often rely on similar structures or internal representations. Thus, even if these evolved functions appear opaque or inefficient to the human observer, they may provide useful inductive biases or contextual cues to learning systems. This approach therefore offers a brige between black-box policy optimization and human-guided reward design, while also inviting a broader perspective on what constitutes an effective and meaningful reward signal.

Table 3.1: A list of crossover and mutation operators for genetic programming, adapted from [71]. Highlighted options were used in the experiments conducted in this thesis, further discussed in Chapter 5.

Operation	Possible approaches
Crossover	**Subtree Exchange Crossover**: exchange subtrees between individuals. Self Crossover: exchange subtrees within an individual. Module Crossover: exchange modules between individuals. SCPC: exchange subtrees if coordinates match exactly. WCPC: exchange subtrees if coordinates match approximately.
Mutation	Point Mutation: change the value of a node. Permutation: change the argument order of a node. Hoist: use a subtree to become a new individual. Expansion Mutation: exchange a subtree against a terminal node. Collapse Subtree Mutation: exchange a terminal node against a subtree. **Subtree Mutation**: replace a subtree by another subtree. Gene Duplication: replace a subtree by a terminal.

3.3 Evolutionary Computation and Reinforcement Learning

Despite the success of reinforcement learning (RL) in various domains, it faces significant challenges that stem from its nature. Evolutionary computation offers complementary approaches to address some of these challenges. EC has demonstrated its utility both as a standalone alternative to RL and as a synergistic component within RL frameworks. By evolving populations of policies or value functions, EC bypasses the need for gradient-based optimization, making it particularly robust in non-differentiable or noisy environments. Furthermore, EC is inherently parallelizable, enabling scalable exploration of policy space. Within this context, *evolutionary reinforcement learning* [7], the application of evolutionary computation to RL, has emerged as a family of approaches that integrate learning during the lifetime of an agent with population-wide adaptation through natural selection. This paradigm extends beyond the development of behavioral agents to include the co-evolution of environments and interactions, allowing both agents and their surroundings to adapt dynamically in pursuit of optimal outcomes. The schema representing the cooperation of EC and RL can be seen in Figure 3.3.

This section explores the bridge between evolutionary computation and reinforcement learning. We begin by discussing how EC can be integrated into RL pipelines, providing auxiliary mechanisms for tasks like reward shaping, policy optimization, and hyperparameter tuning. Next, we highlight the use of evolution strategies (ES) as an alternative to traditional RL methods, demonstrating their applicability and advantages in solving complex RL problems. These perspectives lay the foundation for the evolutionary approach to reward design proposed in this thesis.

3.3.1 Evolutionary Computation as a Component in RL

In the following, we present a brief overview of current approaches to integrating EC into RL [48, 6, 65]. This adaptation has demonstrated promising performance in addressing

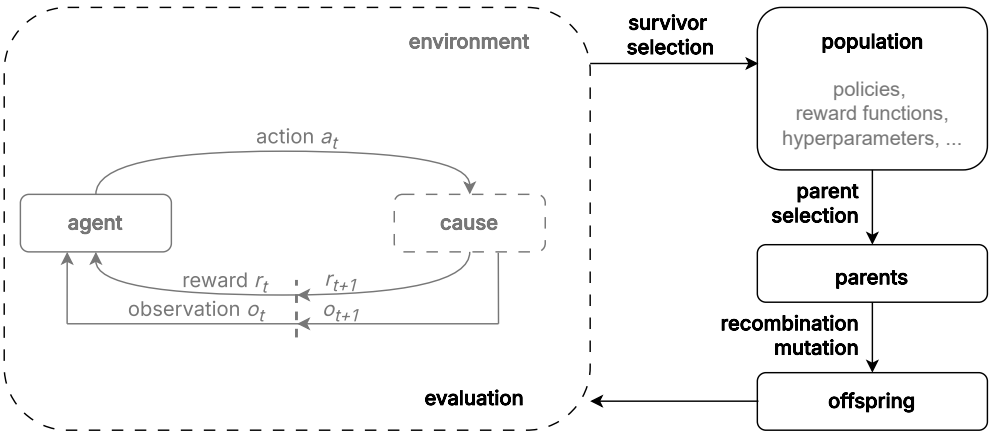


Figure 3.3: **Overview of EC applied in RL.** In the inner loop, an individual agent learns by interacting with its environment. In the outer loop, the evolutionary cycle of selection and variation operate on a population of individuals. Any part of the system might be subject to evolution, including decision policies, reward functions, environment transition functions, initial states, hyperparameters, etc. Image inspired by [7].

limitations such as those stemming from the sensitivity to hyperparameter initialization, a lack of diverse exploration, and conflicting objectives for rewards. We give particular attention to evolutionary computation in reward shaping, as it is the main focus of this thesis.

EC in Hyperparameter Optimization. Identifying the optimal hyperparameter setting for RL is a challenging task due to a large number of hyperparameters related to the RL algorithm itself, as well as those needed for the neural network architectures of policies. Hyperparameter optimization for RL faces several problems as the performance evaluation can be extremely expensive for complex tasks, the search space of hyperparameters can be complex, involving high dimensionality. In addition, there might be multiple contradictory objectives that need to be optimized. Due to the high degree of parallelism, their gradient-free properties and the ability to obtain a set of trade-off optimal solutions, the EC-based methods have shown potential to meet these challenges [47, 33]. Evolutionary methods typically involve random initialization of parameters while evolving hyperparameters via genetic algorithms [27, 26]. In *population-based training* [11], parameters are inherited while the hyperparameters are adapted to the current learning process to make the agents learn more efficiently. The combination of these two approaches has also been explored in [28].

EC in Policy Search. Following the popularization of deep learning, the incorporation of neural networks (NNs) as function approximators for policies has been widely popular. *Neuroevolution* has emerged as an alternative to the classical stochastic gradient descent approach. This method leverages gradient-free EC methods for policy search, which can optimize NN weights, architectures, hyperparameters, building blocks, and even learning rules [94, 35, 95].

EC in Exploration. The exploration of the environment introduces multiple challenges such as large state space, environment that returns sparse and delayed rewards,

presence of noise which makes it difficult to observe only relevant information, and introduction of multiple agents, which makes the state action space exponentially increase. EC methods enable extreme exploration, competition, and cooperation, in addition to massive parallelization by maintaining a set of diverse agents during search. Evolutionary exploration is introduced in two approaches, by *diversity encouragement methods* [57, 75, 37], especially for neuroevolution, and *evolution-guided exploration* [15, 60, 59] methods for traditional RL algorithms.

EC in Reward Shaping. *Reward shaping* has been introduced to enhance the original reward with additional shaping rewards. The purpose of these is to provide feedback on the progress of the task or adjust the importance of different subproblems. Empirical studies have shown that reward shaping can reduce the amount of exploration and accelerate convergence [55]. However, incorrect reward shaping can alter the problem itself. Achieving a balance between multiple subrewards also requires careful manual tuning. Evolutionary computation has been applied to deal with the aforementioned challenges by directly introducing reward functions. *Potential-based* reward shaping is a method originating from the idea of potential energy. It guarantees consistency with optimal policy [68]. The optimal reward function can be achieved by maximizing the expected fitness over the distribution of environments, in a standard approach using general evolutionary computational framework [91]. Another method, *PushGP* [70] can efficiently find reward functions discovering common features of environments. This thesis focuses on using genetic programming for reward shaping. Further details and the proposed method is discussed in Chapter 5.

3.3.2 Evolutionary Computation as an Alternative Approach to RL

A lot of effort has been put towards creation of beahvioral agents using evolutionary computation. Early work by Holland [42] and Koza [52] describes methods of evolving game-playing strategies and controllers for complex dynamical systems. These foundational studies paved the way for modern approaches to leveraging EC in reinforcement learning. Evolution Strategies (ES), a class of optimization techniques within EC, has gained attention as a black-box optimization approach to solving RL problems. Unlike traditional RL methods that rely on the Markov Decision Process (MDP) framework, ES directly optimizes policies by evaluating their performance as a whole. This distinction gives ES several advantages, particularly in settings with deceptive or sparse rewards. There is *no gradient backpropagation*, as ES avoids the gradient calculations used in traditional RL, making it robust to nondifferentiable or noisy environments. ES also bypasses the need to estimate value functions, simplifying the optimization process. Furthermore, ES handles delayed or long-term rewards without requiring specific temporal credit assignment mechanisms. These characteristics make ES fast, scalable, and easy to train, as demonstrated by Salimans et al. [81], who successfully applied ES to various RL benchmarks.

While RL and EC share similarities, such as the exploration of policy space, their approaches to optimization are significantly different. Traditional RL relies heavily on the use of value functions for efficient search, while EC methods perform direct optimization in the policy space guided by scalar evaluations of entire policies. Despite these differences, EC and RL are not merely alternatives but can also complement each other. The emerging field of evolutionary reinforcement learning leverages this synergy by using RL to enhance EC methods and vice versa, creating hybrid approaches that capitalize on the strengths of both paradigms.

Chapter 4

Reward Engineering and Shaping

Successful training of RL agent on complex domain requires a large amount of experience stemming from ongoing interaction with the environment. Based on experiments from the field of behavioral psychology, which involved monitoring of animal's change in dopamine levels in response to being rewarded for carrying out a certain action [93], such mechanism has been translated into the computational domain. Reward function is the main source of the environment's response to artificial agent's behavior, directly evaluating whether its behavior matches the intended goal. Given that such response coming from the environment is the core mechanism driving agent's learning, proper setting of environment's feedback signal is crucial for learnability of the task at hand. *Reward engineering* is a domain which refers to the broader task of designing, modifying, or generating reward functions to improve the learning dynamics of RL agents. It encompasses both the manual design of reward signals and algorithmic methods that assist or automate this process. In an ideal case, the reward signal provides sufficient feedback to align the agent's behavior with the intended task objectives. However, in practice, designing an effective reward function is rarely straightforward. Improper reward design can lead to negatively unexpected trajectories and even unintended, indirect support of misaligned behavior.

Challenges in Reward Design. Reward engineering involves crafting a signal that aligns the agent's learning incentives with the desired task outcomes. While seemingly straightforward, this process is notoriously difficult in practice. The core challenges and examples of negative side effects that stem from incorrect reward design include: ***Reward Sparsity*** — feedback may occur after long sequences of actions, which can result in slow and unstable learning. ***Reward Function Complexity*** — reward functions with multiple factors can be difficult to design and interpret, and it can be difficult to objectively evaluate the effectiveness of a reward function, especially in complex environments, ***Deceptive Rewards*** — such signals may encourage the agent to find „easy“ solutions that are not aligned with the true objective, meaning that the agent finds locally optimal but globally poor strategies that game the reward function. ***Reward Hacking*** — Agents can exploit unintended loopholes in the reward function to achieve high rewards without fulfilling the desired goal. ***Inefficient and Unstable Learning*** — similarly to the sparsity problem, improper design can have low value for agent's improvement, leading to slow learning, and appearance of negative phenomena such as catastrophic forgetting or oscillatory policy updates during learning. Incorrectly set up rewards can encourage *misaligned behavior*, which results in agent disregarding the original intentions and behaving in ways which do

not match them. In critical settings, it is important to ensure that the agent's behavior complies with the designed task and does not result in unintended behavior.

Following the findings in animal training and behavioral psychology, intermediate reinforcements in the form of direct neural stimulation were shown to be used to gradually teach complex behaviors faster. The provision of supplementary rewards in training was shown to make the problem easier to learn [72]. In RL, this often means introducing additional incentives or penalties that help the agent discover useful behaviors faster. For example, an agent solving a maze might receive small rewards for moving closer to the goal, not just when the goal is reached. This approach of using intermediate rewards that „nudge" the learning algorithm toward more positive actions is called *reward shaping*, with introduced functions being referred to as *shaping (reward) functions*.

Reward shaping has been explored as a method to mitigate catastrophic forgetting in reinforcement learning, to improve performance, mainly by stabilizing training and reducing computational effort. In addition, shaping functions allow for faster convergence compared to unshaped models, and allow enhanced robustness to hyperparameters which is useful in applications of transfer learning. The method's ability to reduce policy oscillation and improve sample efficiency indirectly addresses undesired behavior such as catastrophic forgetting by promoting consistent learning trajectories, functioning as a form of regularization in learning. The formal basis for reward shaping has been established by Laud in [56]. Laud's theory suggests that using shaping approach reduces the *reward horizon*, i.e., the delay between an action and value estimate, thus enabling faster learning.

Reward Horizon. The reward horizon refers to the temporal distance between the agent's action and the feedback it receives. Essentially, it captures the quality of the feedback of a task, and indirectly describes the rate at which the process as a whole can be learned. Long horizons pose challenges for credit assignment, making it difficult for the agent to attribute outcomes to specific actions. Reward shaping mitigates this by offering intermediate feedback, effectively shortening the horizon and speeding up learning. Even though the reward horizon is a conceptual idea rather than a formally defined metric, it summarizes one of the key benefits of using tailored shaping functions, and it can be used as a heuristic to evaluate quality of task's feedback structure, and guide the design of reward functions.

This thesis explores the use of genetic programming in enabling quality reward shaping and design. Before introducing the proposed method in detail, this chapter focuses on introduction of reward engineering and shaping in a broader context.

4.1 Expanding on Reward Functions

When considering reward design and shaping, several foundational concepts are relevant:

Scalar vs. Vector Reward. *Scalar rewards* are the most common form of reward, providing a single numerical signal as a feedback. This approach is simple, easy to interpret, and computationally efficient. However, its nature often leads to the failure to capture the nuances of complex tasks, which can lead to suboptimal behavior.

Vector rewards represent feedback in a multidimensional signal, where each component addresses a different aspect of the task. Such richer feedback offers a more comprehensive evaluation of the agent's actions, gives space to a finer-grained learning process, and enables multiobjective learning. However, this approach introduces complexity in balancing priorities and interpreting results.

Intrinsic vs. Extrinsic Reward. In psychology, experts differentiate between extrinsic motivation, where actions are driven by specific anticipated rewards, and intrinsic motivation, where actions are motivated by inherent enjoyment [80]. In RL, rewards can be categorized in this direction, so that *external rewards* are tied directly to the environment (e.g., score in a game), and *intrinsic rewards*, which reflect internal motivations, such as curiosity, exploration bonuses, or reduction in uncertainty. The intrinsic approach is often used to improve exploration in sparse environments.

4.2 Categories of Reward Shaping

Reward shaping can be categorized along several dimensions, based on underlying principles of a given approach. Examples of such categories include:

- **Static shaping.** In this approach, the reward function is fixed throughout the training process. Such type of shaping is easy to implement, but may not adapt well to changing environment, task dynamics or agent behavior.
- **Dynamic shaping.** Dynamic approach involves modification of the reward function over time based on the agent's performance, or environment dynamics. This approach can offer more targeted and controlled guidance, but requires more complex management and establishment process.
- **Automatic reward shaping.** Automatic design aims to optimize the reward function itself, without need for its manual crafting. Methods such as genetic algorithms can be used to evolve the most effective shaping functions based on predefined fitness criteria. This approach is explored in this thesis, as it allows to avoid the issue of human bias and domain-specific knowledge limitations, as it allows the reward function to evolve autonomously.

Each approach has trade-offs between flexibility, stability, and implementation complexity. Among these approaches, some methods are additionally guided by formal guarantees that ensure desirable properties, such as policy invariance. One of the most influential and widely studied frameworks in this direction is potential-based reward shaping. Note that some sources argue that potential-based approach is the only formally valid approach to shaping, as it uniquely guarantees policy invariance under certain conditions [68]. Others take a more relaxed view, treating shaping as any additional reward signal that can guide learning, regardless of formal guarantees [38, 40, 18].

4.2.1 Potential-Based Reward Shaping

Potential-based reward shaping (PBRS) [69] is an approach where a potential function $\phi(s)$ is introduced to modify the rewards. The shaping function $R'(s, a, s')$ is constrained to the following form:

$$\mathcal{R}'^{a}_{ss'} = \mathcal{R}^{a}_{ss'} + \gamma\phi(s') - \phi(s).$$

In this approach, the shaped reward $\mathcal{R}'^{a}_{ss'}$ guarantees that the optimal policy is preserved. This means that adding such form of shaping function does not change what the *best actions* are, it only affects how quickly the agent learns them. Traditionally, the potential function $\phi(s)$ was designed manually on the basis of expert knowledge. However, modern approaches use neural networks to learn $\phi(s)$. The neural network-based approach is often applied in the context of transfer learning [5]. There, knowledge from previously solved tasks is reused to guide learning in new environments. These methods often extract potential functions from cumulative reward traces or value functions learned in source domains.

4.3 Beyond Potential-Based Shaping: Motivation for Evolutionary Reward Design

While potential-based reward shaping is theoretically appealing due to its policy invariance guarantees, it has major practical limitations. It relies on hand-crafted or learned shaping functions, which often require careful tuning, good feature representations, or pre-training. It also assumes that preserving the optimal policy is desirable or even necessary, which may not always be the case, particularly in non-stationary or catastrophic forgetting-prone environments.

In this thesis, we adopt a fundamentally different perspective: we do not aim to preserve the original policy, and we deliberately avoid any dependence on human-designed shaping terms. Instead, we propose to automatically evolve the entire shaping function using genetic programming. This method explores a space of symbolic expressions built from basic mathematical operations and the variables provided by the environment's observation space. Crucially, the quality of each shaping function is assessed solely by a fitness function that reflects the desired training behavior, such as fast convergence, stable learning dynamics, and minimal catastrophic forgetting. This allows us to decouple the process of shaping from explicit task modeling: we do not need to encode the task objective into the shaping function directly, as long as our fitness captures the right learning outcomes.

By transferring the burden of reward design from the human to an evolutionary process guided by learning metrics, we potentially:

- Avoid misspecification of rewards and unintended biases.
- Enable discovery of non-obvious shaping strategies.
- Focus on learning dynamics rather than policy preservation.

In summary, this approach generalizes the idea of reward shaping beyond potential-based methods and offers a flexible, domain-agnostic framework for improving RL training in complex or continual settings. In the following chapter, we introduce the proposed method in detail.

Chapter 5

Proposed Method

This thesis focuses on the use of evolutionary computation for *reward shaping* in reinforcement learning. Reward shaping refers to the process of modifying or extending the environment's reward function to better guide the agent toward desired behaviors, promote faster convergence, and improve the stability of the training. Since translating high-level behavioral intentions into meaningful numerical reward signals is often non-trivial, we propose a method that automates this process through the use of genetic programming. Specifically, the GP method evolves mathematical expressions to serve as *static, scalar* and *extrinsic* shaping functions that are directly added to the original reward function of a given environment. An overview of this approach and its effect on training quality is illustrated in Figure 5.1.

In contrast to potential-based reward shaping discussed in Section 4.2.1, which requires domain-specific prior knowledge and carefully constructed potential functions to ensure policy invariance, our approach is intentionally more general. Rather than constraining the shaping function to a difference of potentials, we allow GP to evolve arbitrary symbolic expressions over the observations and actions of the agent. This grants the evolutionary process the flexibility to discover novel shaping functions, at the cost of potentially altering the optimal policy. However, **our goal is not to preserve exact optimality, but to evaluate whether such evolved shaping functions can support faster learning, reduce catastrophic forgetting, and improve adaptation in transfer learning settings.**

By reframing reward shaping as a fitness optimization problem, the burden of manual reward engineering is shifted toward the design of appropriate fitness functions. These fitness functions are designed to promote smooth and stable learning dynamics, rapid convergence, and the absence of known failure modes, such as reward hacking or catastrophic forgetting. The result is a fully automatic pipeline for evolving scalar shaping functions, which can be evaluated across various tasks and environments without requiring manual, domain-specific design. The proposed framework is structured around a fixed RL agent architecture and a training algorithm, which remains unchanged throughout the evolution process. In practice, these agents are configured with predefined hyperparameters, rather than undergoing extensive fine-tuning. The genetic programming algorithm operates in an outer loop, where each individual in the population represents a candidate shaping function. To evaluate an individual, the corresponding reward function is inserted into the environment, and the agent is trained using this modified reward signal. The training trace is logged, and a fitness function is applied to assess the quality of the resulting behavior, for example, by measuring cumulative reward, learning speed, or forgetting metrics.

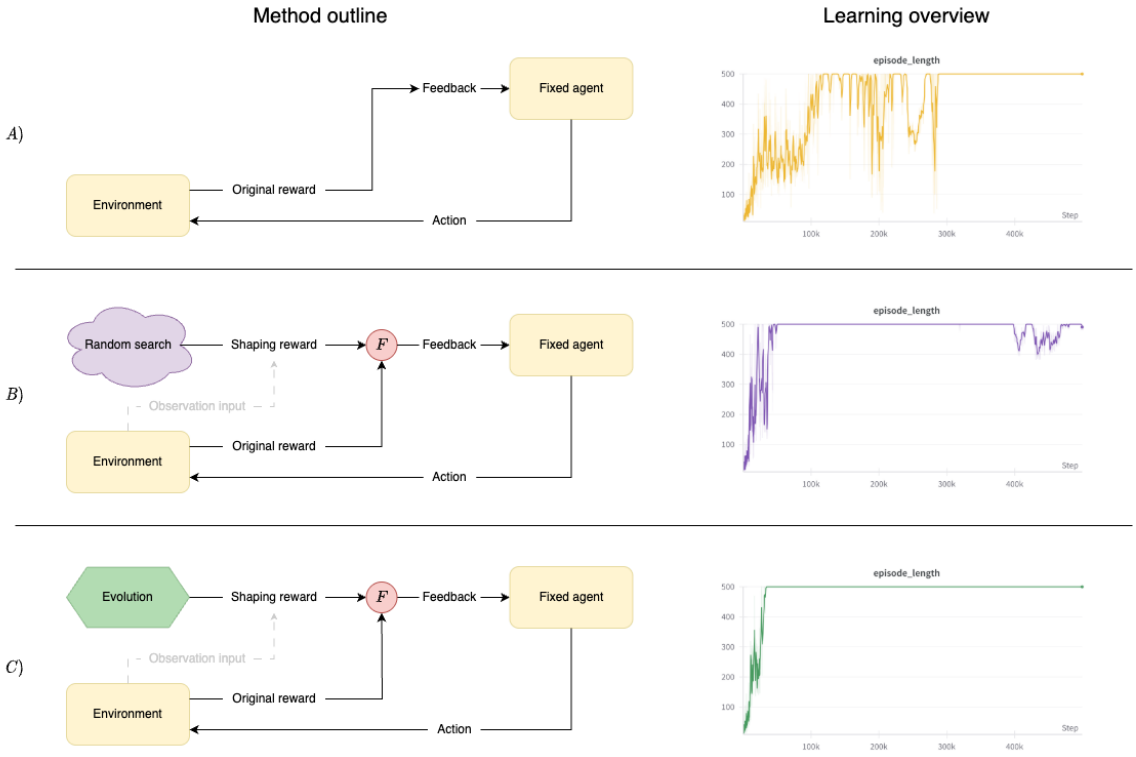


Figure 5.1: **High-level summary of the experiments.** The diagrams on the left illustrate the methodological outline, while the plots on the right show the training quality in terms of episode length. The ideal outcome is fast and stable convergence, with no drop from the maximum performance once reached. *A*) depicts baseline training with no reward shaping, *B*) represents training with a randomly generated reward, and *C*) illustrates training using a shaping reward evolved through evolutionary search. Note that F represents an operation for combining the original reward with the shaping one. In setting of this thesis, addition is used.

After a full evolutionary run, the best performing shaping function is extracted and analyzed. We examine how different evolutionary configurations affect the outcome and compare the shaped learning process with various baselines, including unshaped agents, random search for shaping functions, and conventional shaping heuristics. In additional experiments, we test the transferability of the evolved reward functions by reapplying them to modified versions of the same environment, in which the dynamics have been altered. An overview of the full evolutionary reward shaping process is presented in Figure 5.2, which illustrates the interaction between the genetic programming loop, the RL training environment, and the fitness evaluation process.

5.1 Experimental Setup: Overview

In order to explore the viability of using evolved reward shaping functions for mitigating catastrophic forgetting in reinforcement learning, the following set of experiments was designed to address three main angles: (1) tuning the evolutionary algorithm itself, (2) com-

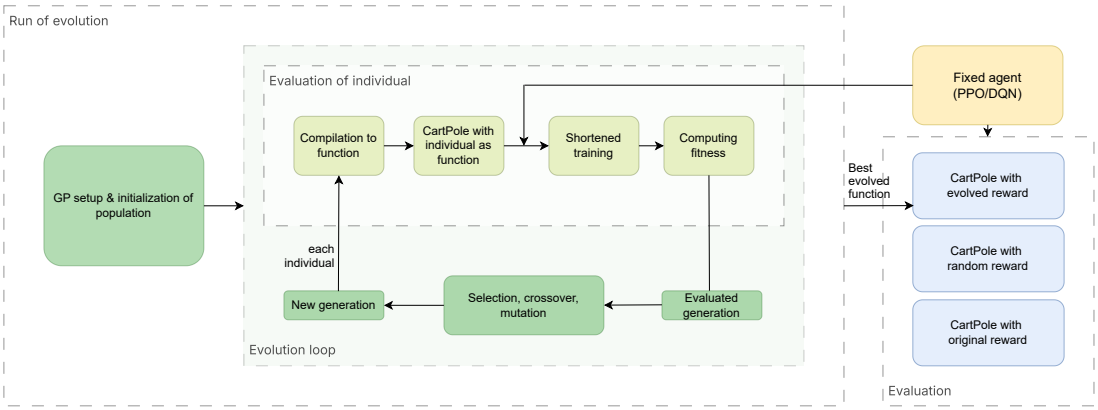


Figure 5.2: Diagram depicting the proposed method.

paring evolved shaping functions to various baselines, and (3) testing generalization through transfer learning.

5.1.1 Tuning Evolution

The first batch of experiments focused on tuning the evolutionary search process. Here, we explore how different population sizes, mutation rates, and tournament selection parameters affect the quality and stability of evolved reward functions. Since each evaluation of a reward function involves training an agent from scratch, the number of training steps was shortened with a reduced training budget (100,000 steps) to keep the overall evolution computationally feasible. Despite reduced training, relative differences in performance and forgetting were still visible, allowing the evolutionary process to make meaningful progress.

5.1.2 Approaches to Obtaining Shaping Function

To evaluate the impact of different reward shaping strategies, we designed a comparative experiment using multiple approaches. Once a reasonable evolutionary setup was in place, the following experiments using four types of reward shaping strategies were run:

- **Evolved**: reward functions produced via genetic programming.
- **Manual**: manually designed shaping functions based on known heuristics (e.g., cart velocity or pole angle).
- **Random**: syntactically valid but randomly generated reward functions.
- **None**: the default environment reward (i.e., no shaping).

These variants were tested on two RL algorithms: PPO and DQN on the CartPole environment. This allowed for examining whether evolved shaping functions offer meaningful benefits in reducing catastrophic forgetting compared to more naïve or traditional alternatives. All agents were trained under a consistent training schedule and evaluated using forgetting-aware fitness metrics described in Section 5.2.3.

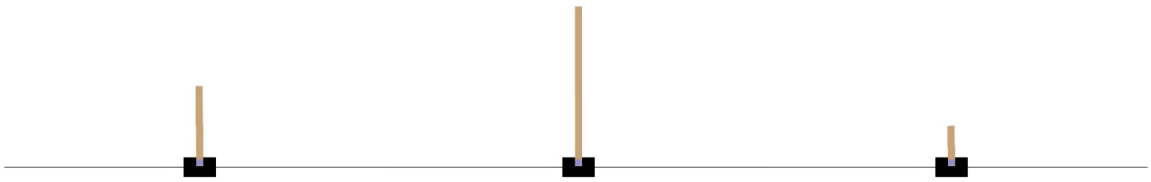


Figure 5.3: **Example of environment variations used in transfer learning experiments.** The figure illustrates three instances of the CartPole environment with differing pole lengths. By modifying a single physical parameter, we test the agent's ability to generalize or transfer learned behavior across related but different tasks.

5.1.3 Transfer Learning

Finally, to test the generality of the evolved reward shaping functions, a set of transfer learning experiments was conducted. In these experiments, shaping functions evolved on the original CartPole environment were reused in modified versions of the same task (e.g., with altered pole length or mass). The goal was to evaluate whether the evolved reward structure could facilitate effective learning in new dynamics without requiring re-evolution, or task-specific tuning.

To isolate the influence of the reward function itself, we adopted a **full retraining** strategy. This means that for each modified environment, the RL agent was re-trained from scratch, with randomly reinitialized weights. No fine-tuning or continual learning was involved. This setup emphasizes the reward function's potential to generalize across task variants independently of policy reuse.

We hypothesize that a well-shaped reward function provides more informative feedback than the original task reward and is therefore more robust to changes in environment dynamics. In general, adapting an RL agent to a new environment or task requires retuning hyperparameters or retraining. If the evolved function is sufficiently general, it may reduce or eliminate the need for such adjustments by consistently guiding the agent towards useful behavior, even in altered setting.

Environment Variants. Modifications listed in Table 5.1 were made to the original CartPole environment to create new task variants. This set of tasks allows us to assess the *cross-environment applicability* of evolved shaping functions. The results are compared with the baseline reward setups to determine whether shaping provides a consistent advantage in altered dynamics. Examples of different pole lengths used in the experiment are shown in Figure 5.3.

5.2 Experimental Setup: Details

In the following, details of the proposed approach are discussed, including types of environment, the architecture and hyperparameters of the reinforcement learning algorithm and the design of the genetic programming framework for evolving reward functions.

Table 5.1: List of environment variants used in the transfer learning experiments, with modified physical parameters.

Environment Variant	**Modified Parameter**	**Default** [31]
CartPole-ShortPole	Pole length = 0.25 (shortened)	0.5
CartPole-LongPole	Pole length = 1.0 (lengthened)	0.5
CartPole-HeavyPole	Pole mass = 0.2 (increased)	0.1
CartPole-LightPole	Pole mass = 0.02 (decreased)	0.1
CartPole-HeavyCart	Cart mass = 2.0 (increased)	1.0
CartPole-LowGravity	Gravity = 4.9 (reduced)	9.8
CartPole-HighGravity	Gravity = 19.6 (increased)	9.8

5.2.1 Baseline

To assess the effectiveness of the evolved reward shaping functions, we compare them against a set of baselines. These baselines provide reference points that help contextualize performance gains and ensure that improvements are not due to chance or overfitting. Each baseline is designed to isolate different aspects of reward design or agent behavior, including unshaped learning, random shaping, and manually constructed shaping functions.

Original Environment

For both PPO and DQN agents, we define a baseline that uses the environment's default reward function without any shaping. In the case of CartPole, this consists of a constant reward of +1 for every timestep in which the pole remains balanced. This serves as a control condition to evaluate whether the addition of any shaping, whether evolved, manually designed, or randomly generated, produces measurable differences in performance or forgetting. Further details of the environment are provided in Section 5.2.2.

Random Search

To evaluate whether the computational cost of the evolutionary process is justified, we include a random search baseline in our experiments. Random search is often used as a reference point in black-box optimization, offering a simple yet surprisingly competitive baseline in many settings. In our case, random search involves sampling a fixed number of syntactically valid reward expressions from the same expression space defined for genetic programming. These expressions are generated without applying any selection, crossover, or mutation, ensuring that the process remains entirely unguided.

Each sampled expression is compiled into a function and evaluated using the same training loop and fitness metrics as used in the evolutionary setup. The best-performing reward shaping function from this pool is retained and used for comparison. The number of random samples is matched to the total number of evaluations used in the evolutionary setup, to ensure a fair comparison in terms of computational budget.

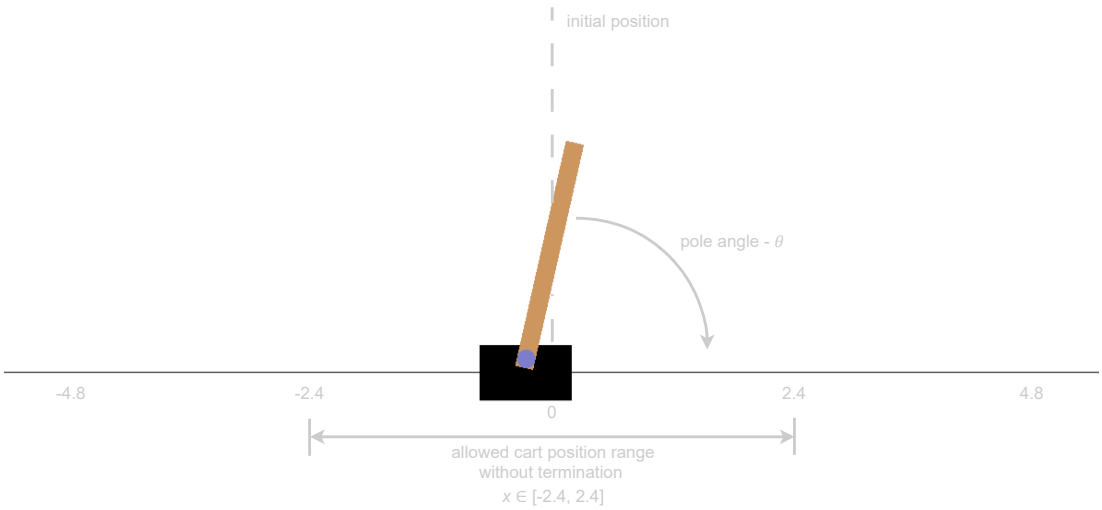


Figure 5.4: **Example configuration of CartPole environment in its base setting.** The environment's state consists of four variables: cart position x, cart velocity v, pole angle θ, and pole angular velocity ω, as annotated in the figure.

5.2.2 Environment

The main environment used throughout the experiments is `CartPole-v1`, a classic control task included in the Gymnasium library [32]. It involves a pole attached to a cart that moves along a frictionless track. The agent must learn to apply forces (left or right) to keep the pole balanced upright for as long as possible. Visualisation of this task is shown in Figure 5.4.

Although the physics behind the system is quite simple and could easily be modeled using classical control theory (e.g., via proportional–integral–derivative (PID) controllers [29]), the experiments in this thesis use a model-free reinforcement learning approach. The agent has no prior knowledge of the dynamics of the system or the semantics of the observation values: it must learn purely by interacting with the environment and receiving scalar reward feedback. Unlike PID controllers, which require manual tuning and accurate system modeling, reinforcement learning can adapt to complex, nonlinear, or partially unknown dynamics and potentially generalize better to variations or disturbances.

Action space. The action space is discrete, with two possible actions:

- `0`: push the cart to the left,
- `1`: push the cart to the right.

Observation space. The observation space is a continuous, four-dimensional space, where each observation is a vector (x, v, θ, ω) consisting of:

- x denoting `cart position`,
- v denoting `cart velocity`,

Table 5.2: Minimum and maximum values for each observation component in the CartPole environment.

Index	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	~ -0.418 rad ($-24°$)	~ 0.418 rad (24°)
3	Pole Angular Velocity	$-\infty$	∞

- θ denoting `pole angle`,
- ω denoting `pole angular velocity`.

All values are unnormalized, and the agent does not receive explicit information about the structure or meaning of the vector components. Table 5.2 contains maximum and minimum possible values of the observation elements[1]. Note that while the ranges denote the possible values for observation space of each element, they are not reflective of the allowed values of the state space in an unterminated episode. In particular, the cart position (index 0) can take values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range. Furthermore, the pole angle can be observed between (-0.418, 0.418) radians (or $\pm 24°$), but the episode terminates if the pole angle is not in the range (-0.2095, 0.2095) radians (or $\pm 12°$).

Reward. Since the goal is to keep the pole upright for as long as possible, by default, the agent receives a reward of $+1$ for every timestep the pole remains upright, including the termination step. The maximum episode length is 500 steps in `CartPole-v1`, so the maximum possible cumulative reward is 500. In this setting, the environment is considered solved when an agent achieves an average score of 475 or higher over 100 consecutive episodes.

Modifications for transfer learning. In experiments involving transfer learning, the environment was extended via wrappers to allow changes to key physical parameters. These include *pole length*, *pole mass*, *cart mass*, and *gravity constant*. These modifications were used to evaluate the generalization ability of agents and the robustness of evolved reward functions across variations of the same base task.

5.2.3 Evolution

This section describes the setup of the evolutionary process used to discover effective reward shaping functions. The search is carried out using genetic programming. Each individual represents a candidate mathematical expression for shaping the base reward signal during PPO training, and evolution terminates only after pre-defined number of evaluations was achieved. The evolutionary process aims to optimize these expressions to reduce catastrophic forgetting while preserving agent performance.

[1] https://github.com/Farama-Foundation/Gymnasium/blob/v0.29.0/gymnasium/envs/classic_control/cartpole.py#L51

Individual representation. Each individual in the population is a tree-structured symbolic expression composed of mathematical operations and input terminals. The terminal set consists of the elements of the environment's observation vector (x, v, θ, ω) and randomly generated constants. The set of functions includes exponentiation, basic arithmetic operators $(+, -, \times, \div)$, and negation, which simply flips the sign of its input. All operations are implemented in their *protected forms* to ensure numerical stability; for example, division is defined as a safe function that adds a small constant to the absolute value of the denominator, preventing divide-by-zero errors. This allows individuals producing near-singular expressions to remain valid and avoids runtime exceptions during evaluation.

Fitness function. During the implementation of this thesis, several versions of fitness functions were explored to effectively guide the evolution of reward shaping functions. Fine-tuning the fitness function was crucial to ensure that the evolutionary process selected individuals that both performed well in the environment and mitigated catastrophic forgetting. The early versions of the fitness function were based primarily on the final performance metrics, such as the average reward obtained during the evaluation episodes. These early versions lacked consideration for catastrophic forgetting, reward volatility, and domain consistency. Subsequent versions incorporated penalties for forgetting (e.g., drops in reward after a peak), volatility (e.g., standard deviation of rewards), and domain consistency (e.g., violations in the reward range). These adjustments allowed for a more balanced and effective evolution of reward functions, which ultimately led to the selection of the hybrid fitness function, which combines reward performance, stability, and forgetting penalties, as the final evaluation method for this experiment.

The final fitness function used for the evolution of reward shaping functions combines several factors to balance performance and generality. At a high level, each candidate shaping function is compiled from a genetic program and used to train a reinforcement learning agent for a reduced number of phases (50). During this warm-up training, the reward shaping function modifies the environment reward, and the agent's performance (episode lengths) is tracked to produce a reward curve. The primary component of fitness is the total reward accumulated during this training period.

To discourage pathological or non-general shaping functions, additional penalties are applied:

- **Domain consistency penalty**: A sample of input states is drawn from the environment's observation space, and the reward function is evaluated on these points. Penalties are assigned if any reward values fall outside the desired $[0, 1]$ range.
- **Variance penalty**: If the shaping function is nearly constant across the input space (variance below a threshold), a strong penalty is applied. This discourages degenerate solutions that return the same reward regardless of the agent's state.

The final fitness is computed as the sum of total episode rewards, minus the weighted domain and variance penalties. Formally, for shaping function f, fitness is defined as:

$$\text{fitness}(f) = \sum_{t=1}^{T} R_t - \lambda_{\text{domain}} \cdot P_{\text{domain}}(f) - \lambda_{\text{const}} \cdot P_{\text{const}}(f),$$

where T is the total number of timesteps in the episode, R_t is the reward at time t, P_{domain} measures the average violation of the $[0, 1]$ reward range, and P_{const} is 1 if the

function is nearly constant, 0 otherwise. This fitness function is maximized during evolution, guiding the search toward shaping functions that both improve learning efficiency and remain valid across a wide range of inputs.

Evaluation of individual. The evaluation of an individual (i.e., a candidate reward function) is performed as follows:

1. **Compilation of the GP tree**: The evolved individual, represented as a GP tree, is compiled into a callable function for integration with the environment.
2. **Unique environment creation**: A custom CartPole environment instance is created with the evolved reward function injected, optionally including reward clipping to improve stability for certain agent types.
3. **Agent training setup**: Depending on the type of agent selected (PPO or DQN), a corresponding trainer class is instantiated to perform a brief training run. The agent's performance is then evaluated using a fitness function that combines reward outcomes with additional metrics such as catastrophic forgetting and stability.
4. **Elapsed time**: The time taken to evaluate each individual is measured, and the result is stored to track the computational efficiency of the evaluation process.
5. **Final reward**: The final fitness score, which is the agent's performance in the environment under the influence of the evolved reward function, is calculated. This score is used to guide the evolutionary process, and individuals resulting in higher fitness scores are favored for reproduction in subsequent generations.

In summary, the evaluation of an individual involves running the agent in a custom, environment-specific setup where the evolved reward function is applied, followed by a brief training run using the selected reinforcement learning agent. This process allows for an objective measure of the reward function's effectiveness in shaping agent behavior, while accounting for important factors like catastrophic forgetting.

Mutation, recombination, selection methods. The evolutionary setup uses standard Genetic Programming (GP) operators from DEAP, with the following details:

- **Mutation**: Subtree mutation is used, where a randomly selected subtree is replaced with a new randomly generated expression. This mutation is performed using protected operations to ensure numerical stability and prevent issues like division by zero.
- **Crossover**: One-point crossover exchanges subtrees between two parent trees, allowing for the recombination of reward shaping functions and the exploration of new combinations of functional components.
- **Selection**: Tournament selection (with size $k \in \{2, 3, 5\}$) is used to select individuals for the next generation based on their fitness, with the tournament size providing a balance between exploration and exploitation.
- **Elitism**: Elitism can be enabled in the selection process using a bool flag, ensuring that the best-performing individuals are carried over directly to the next generation without being altered, preserving good solutions.

In addition to standard evolutionary operators, the experiments support injecting manually crafted individuals into the population to help jump-start the evolutionary process, as well as enabling elitism during a single evolutionary run. These injected individuals correspond to predefined reward shaping functions designed to promote specific agent behaviors or preferences, for example, prioritizing reduced pole angle deviation or encouraging more stable cart movement. Incorporating such domain knowledge can guide the search and potentially accelerate convergence.

The initial population is generated using the `genHalfAndHalf` mode, which produces a mix of *full* method, where all branches are filled up to a specified maximum depth, and the *grow* method, where branches may terminate earlier, resulting in tree of more irregular shapes and varying depth. This approach is commonly used in GP to ensure a diverse initial population, providing a good balance between exploratory and exploitative behaviors in the early stages of the evolutionary process.

The depth of generated trees is restricted to 17, which is a constant recommended by Koza in his book on genetic programming [51]. Such restriction is an implementation of bloat control, to avoid this drawback of genetic programming.

5.3 Implementation Details

All experiments were implemented in Python 3.12, utilizing the `Farama Gym` library (version 0.29.1) [32] for reinforcement learning environments. The PPO and DQN algorithms were implemented from scratch, closely following Callum McDougall's tutorial series [61] on reinforcement learning. No high-level RL libraries like `Stable Baselines3` [78] or `CleanRL` [44] were used, to allow full control over the training loop, agent architectures and logging mechanism. However, the code structure and module layout were inspired by these libraries, with similar organization into separate components for environments, agents, training scripts and utilities.

The reward functions, whether hand-crafted, evolved, or randomly generated, were implemented as symbolic expression trees using the `DEAP` [30] genetic programming library. Each individual in the population represented a reward function encoded as a syntax tree of mathematical operators, state variables, and constants. These trees were compiled into Python functions and injected directly into the environment's reward signal at every timestep using a custom environment wrapper, following the standard interface provided by the Gym API. This environment wrapper also enabled modifications to the environment's physical parameters—such as the length of the pole or the weight of the cart—which were used in the experiments involving transfer learning. This setup allowed for dynamic evaluation and logging of shaped rewards during training.

Experiments were run on the MetaCentrum HPC infrastructure[2], with jobs distributed across available CPU and GPU nodes. All runs were seeded using fixed seeds in the `numpy`, `random`, and `torch` libraries to ensure reproducibility. In experiments focused on the setup of the evolutionary process, multiple seeds were used to average performance and ensure the reliability of results. Each evolutionary setup was evaluated across 20 independent runs using a fixed configuration of parameters. These runs were executed in parallel to speed up the computation process.

The hyperparameters of DQN and PPO algorithms were fixed based on values reported in [43, 76, 77]. These include learning rate, discount factor, entropy coefficients, batch sizes,

[2] https://metavo.metacentrum.cz/

and clipping thresholds, and were kept constant throughout experiments to isolate the effects of reward shaping. A full list of fixed hyperparameters is included in Appendix A. The training loop, logging, and evaluation pipeline were built around `wandb`[3] to track metrics across runs. Code was modularized to support easy integration of different reward functions, whether static, learned, or evolved through genetic programming. The full codebase includes a `requirements.txt` file listing all exact versions of used Python libraries. All training runs were logged using `wandb` to record metrics, track evolving rewards, and compare agent performance across experimental conditions. Additionally, detailed description of the codebase is included in `README.md` file, together with tutorial on how to run the application locally.

[3] https://wandb.ai/site

Chapter 6

Experimental Evaluation

This chapter presents the results of the experiments described in Section 5.1, aimed at evaluating the effectiveness of evolved reward shaping functions in reinforcement learning. The analysis focuses on training performance, robustness, and generalization across different agent configurations and environment dynamics. To improve visual clarity and interpretability of the results, Gaussian kernel smoothing[1] was applied to the plotted curves. Note that this smoothing is used solely for visualization purposes and does not imply statistical significance.

6.1 Training Agents

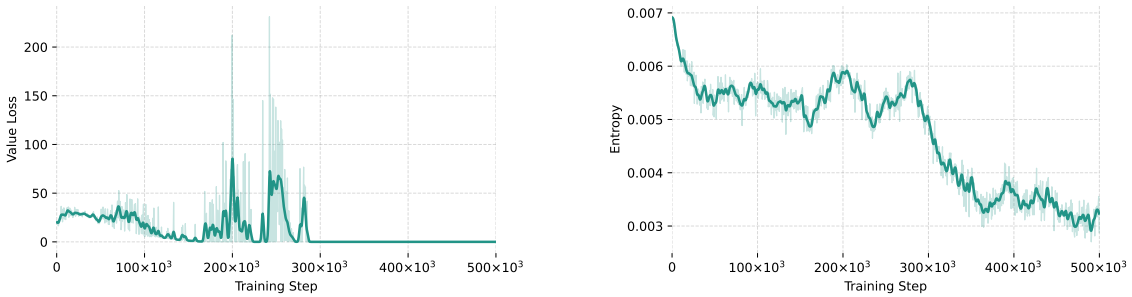
During the training of reinforcement learning agents, multiple metrics were tracked to monitor the convergence of the agent's learning and the stability of its behavior. These metrics are crucial for understanding how well the agent is performing and whether it is learning effectively. The tracked metrics for the PPO and DQN agents are shown in Figure 6.1 and Figure 6.2, respectively. A brief description of each metric is provided below, but for the purpose of evaluating our method and comparing it to other approaches, we focus primarily on the *episode length* metric, as it serves as a direct indicator of agent performance. Specifically, in the CartPole environment, maximizing the duration in which the pole remains balanced is the key objective, and thus the episode length is an important measure of how well the agent is achieving this goal. For a more detailed explanation of the training algorithms and the significance of the tracked metrics (e.g. TD error), see Chapter 2.

6.1.1 PPO Training

The following subset of tracked metrics was used to assess the quality of learning for the PPO agent:

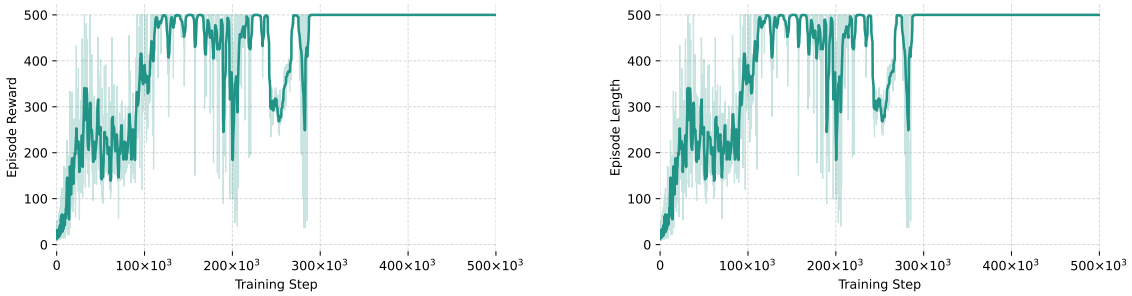
- **Value loss** (Figure 6.1a): This metric represents the difference between the predicted value and the actual returns, which provides insight into the accuracy of the value function. An ideal state would show a decrease in value loss over time as the agent's value estimates converge.

[1] Gaussian smoothing computes a weighted average of neighboring values, where weights follow a Gaussian distribution determined by a chosen standard deviation. It helps reveal trends while preserving the overall shape of the data.



(a) Value Loss

(b) Entropy



(c) Episode Reward

(d) Episode Length

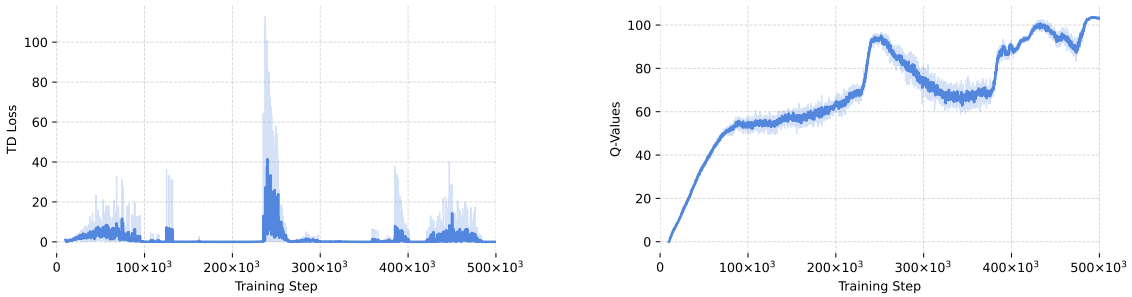
Figure 6.1: **Training metrics for PPO on CartPole.** Value loss and entropy (top row) provide insight into learning stability and policy confidence. Episode reward and episode length (bottom) both reflect the agent's performance, which improves as training progresses.

- **Entropy** (Figure 6.1b): Entropy measures the uncertainty in the agent's policy. Higher entropy indicates greater exploration, while lower entropy reflects greater confidence in action selection. In our implementation, the entropy is computed automatically using `Categorical` distribution and related methods from PyTorch library, which implement the standard discrete entropy formula in a numerically stable way. As training progresses, the entropy is expected to decrease as the agent converges to a more confident policy, without becoming prematurely overconfident.

- **Episode reward** (Figure 6.1c): This metric represents the total reward accumulated by the agent in a single episode. As the agent improves, the episode reward should increase, reflecting a better performance in balancing the pole.

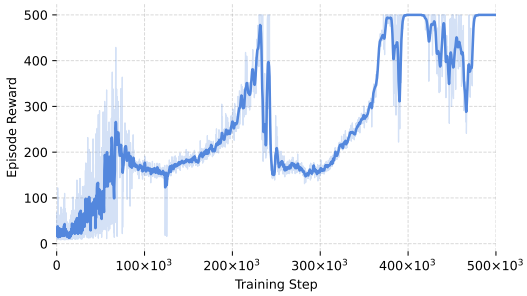
- **Episode length** (Figure 6.1d): In the CartPole setting, this graph shows how long the agent was able to keep the pole upright in each episode. As the agent learns, the values should increase, indicating improved learning of the task.

In the standard CartPole setting without reward shaping, the episode reward and the episode length are effectively equivalent, as the agent receives a reward of $+1$ for each time step the pole remains balanced. Thus, an increase in one directly implies an increase in the other. This equivalence allows both metrics to be interpreted as indicators of the task performance. The metrics presented are crucial for understanding the progress of the PPO agent. The value loss and entropy provide insight into the stability and exploration-

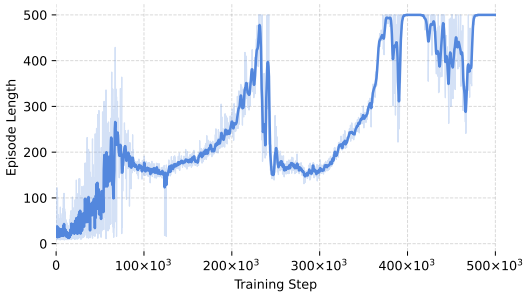


(a) TD Loss

(b) Q-Values



(c) Episode Reward



(d) Episode Length

Figure 6.2: **Training metrics for DQN on CartPole.** TD loss and average Q-values (top row) reflect the accuracy and stability of value estimates, while episode reward and episode length (bottom row) indicate learning progress, with performance improving over time.

exploitation balance of the policy, while the episode reward and episode length directly reflect how well the agent performs the task.

6.1.2 DQN Training

Similarly, for the DQN agent, the following metrics were tracked:

- **TD loss** (Figure 6.2a): Temporal Difference (TD) loss is used to measure the error in the Q-value estimates. A decreasing TD loss indicates that the Q-values are becoming more accurate over time, leading to improved decision-making by the agent.
- **Q-values** (Figure 6.2b): The Q-values represent the agent's estimate of future rewards for each action. An ideal scenario would show stable or increasing Q-values, indicating that the agent is learning good action-value pairs.
- **Episode reward** (Figure 6.2c): As with the PPO agent, the episode reward is a direct indicator of the DQN agent's performance. Higher rewards signal better task execution, where the agent has learned to balance the pole effectively.
- **Episode length** (Figure 6.2d): This metric tracks how long the agent is able to keep the pole balanced, with longer episodes indicating improved learning and performance.

Table 6.1: Listing of hyperparameter settings used in the **A1 experiment**.

Id	Population Size	Number of Generations	Tournament Size	Mutation Probability	Crossover Probability
A1-0	250	30	5	0.3	0.5
A1-1	300	25	5	0.3	0.5
A1-2	500	15	5	0.3	0.5

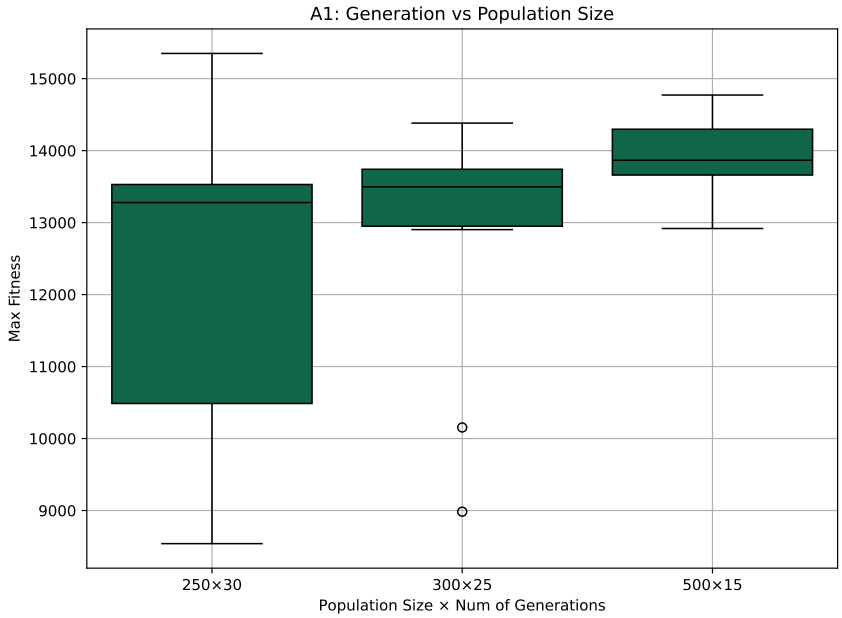


Figure 6.3: Boxplot showing the **distribution of fitness values for different combinations of population size and number of generations** under a fixed total number of evaluations. The x-axis labels represent *population size* × *number of generations*.

For DQN, TD loss and Q-values reflect the accuracy of the value function and the agent's learning process. The episode reward and episode length metrics provide a direct evaluation of the agent's success in achieving the task goal, just like in the PPO case.

6.2 Experiment Dimension A: Tuning for Reward Evolution

Before conducting the main reward evolution experiments, it was critical to tune the evolutionary algorithm's key hyperparameters to ensure robust and meaningful outcomes. Without appropriate tuning, the evolutionary process might suffer from premature convergence, poor exploration, or inconsistent fitness estimation — all of which could mask the true potential of evolved reward functions or distort comparisons between methods.

To address this, we designed a dedicated tuning phase, referred to as **Experiment Dimension A**. This phase focused on evaluating the sensitivity of reward evolution to various evolutionary hyperparameters, including population size, number of generations,

mutation and crossover probabilities, and tournament size. We conducted experiments with a total of 72 unique hyperparameter configurations, evaluated under consistent conditions to maintain comparability. A central constraint across all experiments was a **fixed total number of evaluations** per configuration, defined as:

$$\begin{aligned}\texttt{total\_evals} &= \texttt{population\_size} \times \texttt{num\_generations} \times \texttt{num\_independent\_runs} \\ &= 75{,}000.\end{aligned}$$

In following tuning experiments, 10 independent runs were run for each setting, however, in the final experiment for obtaining shaping reward function, number of runs was increased to 20 in each hyperparameter setting. Due to space constraints and clarity, in this chapter, we present only a selection of focused experiments from this broader search. Each experiment isolates the effect of one or two hyperparameters while keeping the total number of evaluations fixed. This allows us to analyze trade-offs and interactions between design choices in a controlled manner. The configurations used in each experiment are listed in the corresponding tables.

The goal of this tuning process was not only to find optimal parameters for our specific setup but also to understand how evolutionary dynamics behave under different configurations, helping us choose robust defaults for subsequent reward evolution experiments.

6.2.1 Experiment A1 – Population Size vs. Number of Generations

This experiment explores how the trade-off between population size and number of generations affects the outcome of reward evolution, under a fixed total number of evaluations. In evolutionary algorithms, both larger populations and longer runs (more generations) can lead to better solutions. However, given a fixed computational budget, it is necessary to find correct balance between the two. This experiment helps determine whether it is more beneficial to favor exploration through larger populations or deeper optimization through more generations. The mutation rate, the crossover rate, and the number of runs are kept constant to isolate this effect. Exact settings of hyperparameters are shown in Table 6.1.

The boxplots in Figure 6.3 clearly show that increasing the size of the population with a smaller number of generations leads to higher fitness values. The 500×15 configuration (largest population, fewest generations) achieves the highest median fitness and demonstrates more consistent results with a narrower interquartile range. In contrast, 250×30 configuration shows the widest variability and lowest median fitness. This suggests that within a fixed computation budget, investing in larger populations yields better optimization performance than running more generations.

6.2.2 Experiment A2 – Different Mutation Probabilities

In this experiment, we examine the effect of varying mutation probabilities on the performance of reward evolution, with exact settings presented in Table 6.2. Mutation introduces random variation into individuals, enabling exploration of new regions in the search space. However, too much mutation can degrade the search by disrupting promising solutions. We test several mutation rates while keeping other settings fixed to evaluate how sensitive the evolutionary process is to this parameter and to identify a balanced value that supports both exploration and stability.

The experiment shows a positive trend between mutation probability and maximum fitness, as seen in Figure 6.4. As mutation probability increases from 0.2 to 0.4, the median

fitness also gradually increases. The 0.4 setting achieves the highest median fitness, though there is considerable overlap between distributions. This suggests that higher mutation rates provide better exploration of the solution space, likely helping the algorithm to avoid local optima.

Table 6.2: Listing of hyperparameter settings used in the **A2 experiment**.

Id	Population Size	Number of Generations	Tournament Size	Mutation Probability	Crossover Probability
A2-0	500	15	5	0.2	0.5
A2-1	500	15	5	0.3	0.5
A2-2	500	15	5	0.4	0.5

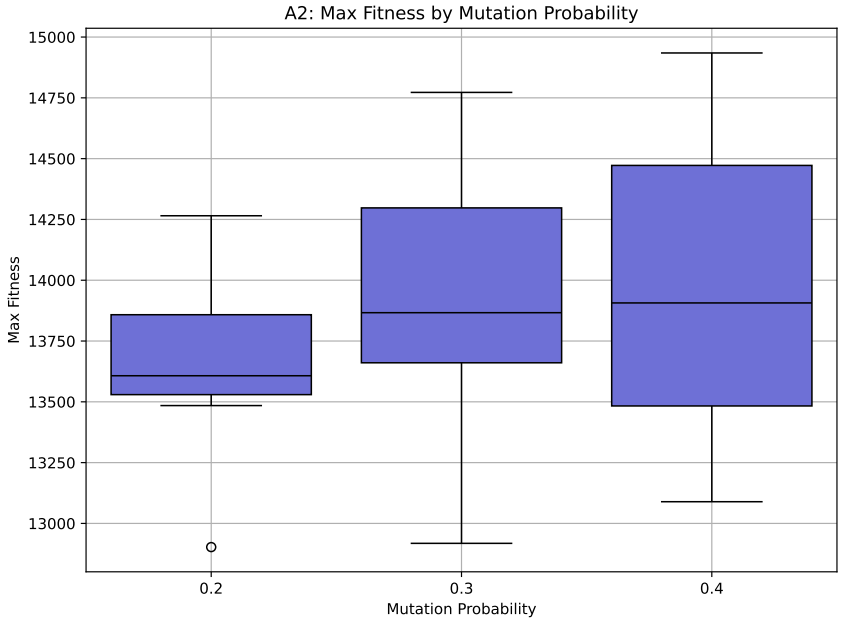


Figure 6.4: Boxplot comparing fitness **outcomes across three mutation probability settings** $(0.2, 0.3, 0.4)$, illustrating the effect of mutation rate on evolutionary algorithm performance.

6.2.3 Experiment A3 – Different Crossover Probabilities

This experiment investigates the impact of different crossover probabilities on the quality and stability of evolved reward functions. Crossover allows information exchange between individuals, potentially accelerating convergence by combining useful building blocks of the individuals. However, high crossover rates may introduce instability or premature convergence if not well-calibrated. We test a range of crossover values under otherwise fixed settings, as presented in Table 6.3, to observe how this parameter influences reward evolution dynamics.

Interestingly, results plotted in Figure 6.5 show that crossover probability exhibits a non-linear relationship with fitness outcomes. The highest setting (0.7) yields the best median

Table 6.3: Listing of hyperparameter settings used in the **A3 experiment**.

Id	Population Size	Number of Generations	Tournament Size	Mutation Probability	Crossover Probability
A2-0	500	15	5	0.3	0.3
A2-1	500	15	5	0.3	0.5
A2-2	500	15	5	0.3	0.7

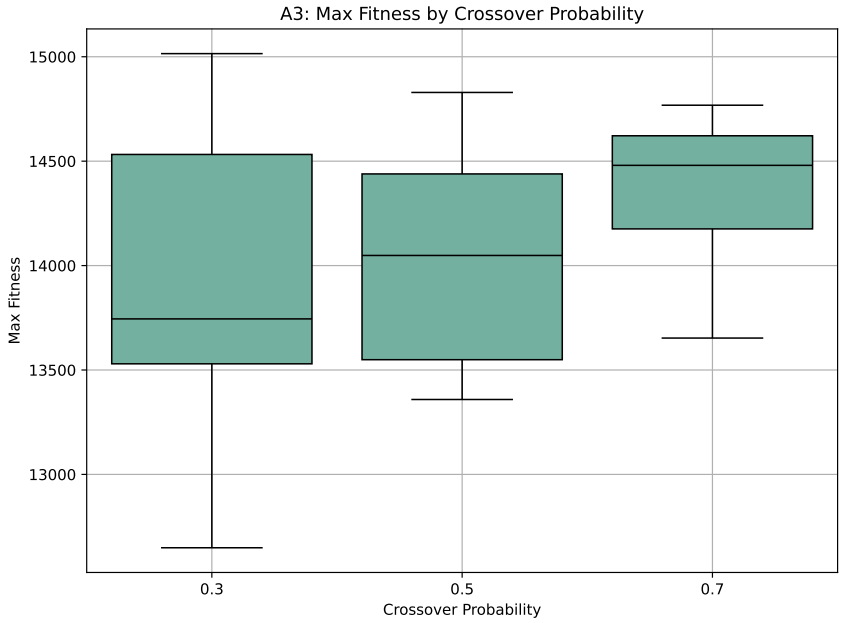


Figure 6.5: Boxplot depicting fitness **distributions for different crossover probabilities** $(0.3, 0.5, 0.7)$, highlighting how crossover rate influences solution quality.

fitness and generally better performance. However, the middle value (0.5) appears to perform worse than the lower setting (0.3), invoking a V-shaped pattern rather than linear improvement. This suggests that while higher crossover rates generally improve performance, there might be interaction effects with other parameters affecting the middle range.

6.2.4 Experiment A4 – Different Tournament Sizes

Here, we evaluate the effect of selection pressure by varying the tournament size used during parent selection. Tournament selection determines how aggressively the algorithm favors individuals with higher fitness. Larger tournaments increase selection pressure, while smaller ones promote diversity. This experiment helps us understand how strong selection influences the trade-off between exploitation and exploration and how it affects the stability and diversity of evolved reward functions. The exact setting of the hyperparameters can be found in Table 6.4.

The results in Figure 6.6 show that tournament size has a clear impact on evolutionary performance, with the size 2 showing a significantly lower mean fitness and much greater

Table 6.4: Listing of hyperparameter settings used in the **A4 experiment**.

ID	Population Size	Number of Generations	Tournament Size	Mutation Probability	Crossover Probability
A4-0	500	15	2	0.3	0.5
A4-1	500	15	5	0.3	0.5
A4-2	500	15	7	0.3	0.5

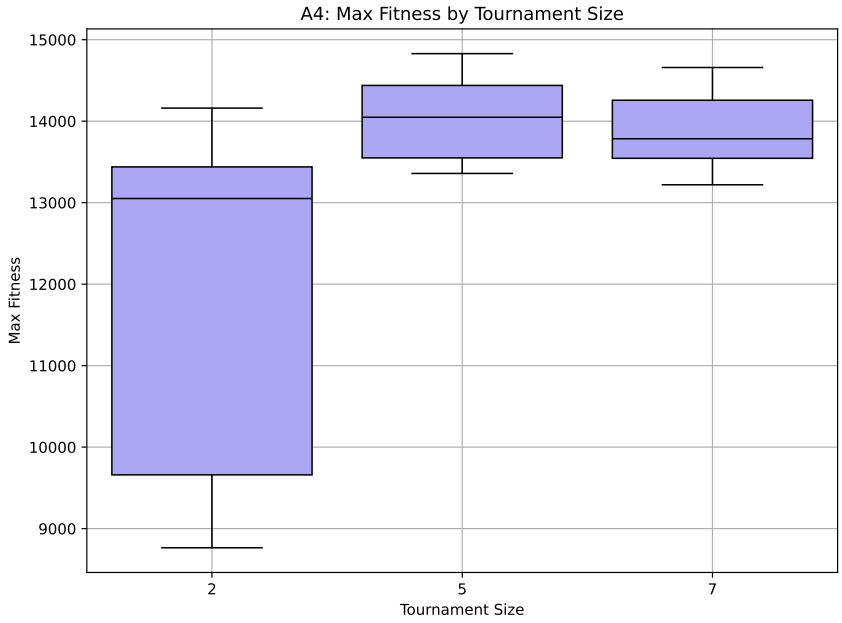


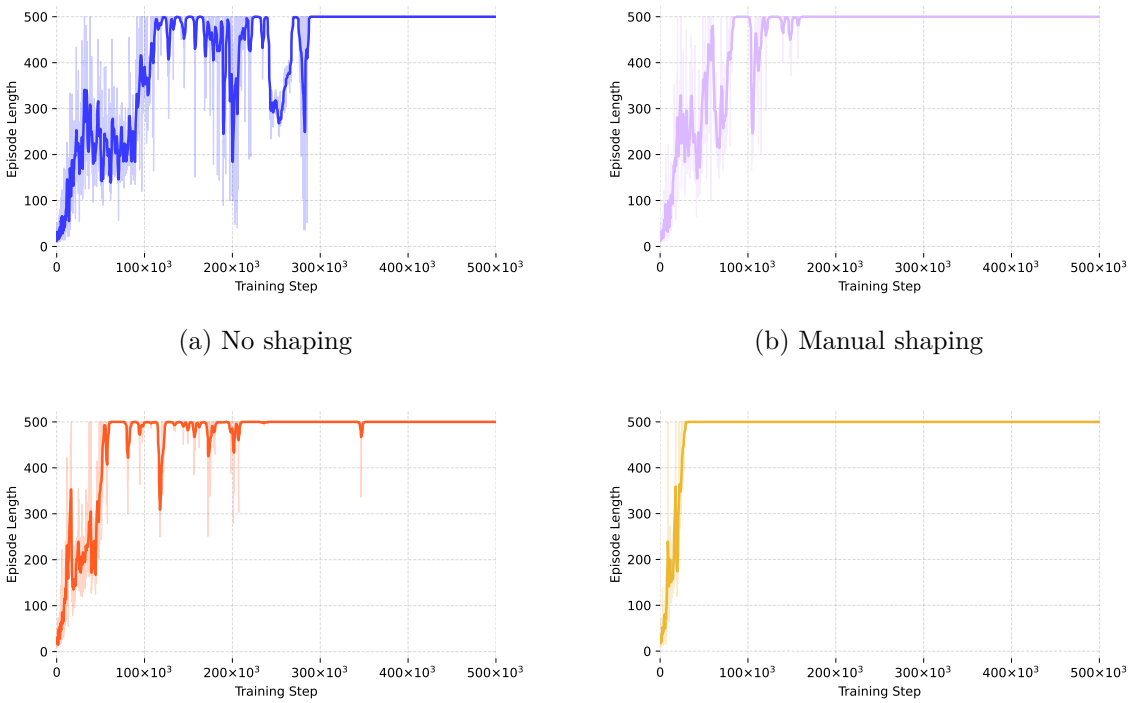
Figure 6.6: Boxplot showing fitness **results for tournament sizes** 2, 5 **and** 7**,** demonstrating the impact of selection pressure on evolutionary optimization.

variability than larger tournament sizes. Both sizes 5 and 7 achieve notably better and more consistent results, with size 5 appearing slightly better than 7 in terms of median fitness. This indicates that very small tournament sizes provide insufficient selection pressure, while very large tournaments might be too elitist. The middle value 5 strikes a good balance between exploration and epxloitation.

6.3 Experiment Dimension B: Comparison of Reward Shaping Approaches

The following experiment focuses on comparison of different ways of obtaining shaping rewards to the baseline with no shaping. In addition to shaping function obtained from process of evolution, we explore shaping functions discovered by random search, and manually crafted one. The random search was run with the number of evaluations equivalent to

$$\texttt{num\_runs} * \texttt{num\_generations} * \texttt{population\_size},$$



(a) No shaping

(b) Manual shaping

(c) Random shaping

(d) Evolved shaping

Figure 6.7: **Comparison of different reward shaping approaches in the PPO (on-policy) agent setting**. Episode length is used as the evaluation metric. Graph (a) depicts a setting with no shaping reward, (b) shows training with a manually-designed reward function, (c) uses a randomly generated function, and (d) uses a reward function evolved through genetic programming.

to keep the comparison equivalent. We allow for random generation of said number of expressions in this case (75,000), and evaluate them using the same approach as calculating fitness in evolution setting. The expression with the highest value is chosen for the comparison. In both PPO and DQN settings, we experimented with a manually designed reward function defined as:

$$1 - \frac{|\theta \div 0.2095| + |x \div 2.4|}{2}$$

which in the genetic programming representation corresponds to the expression

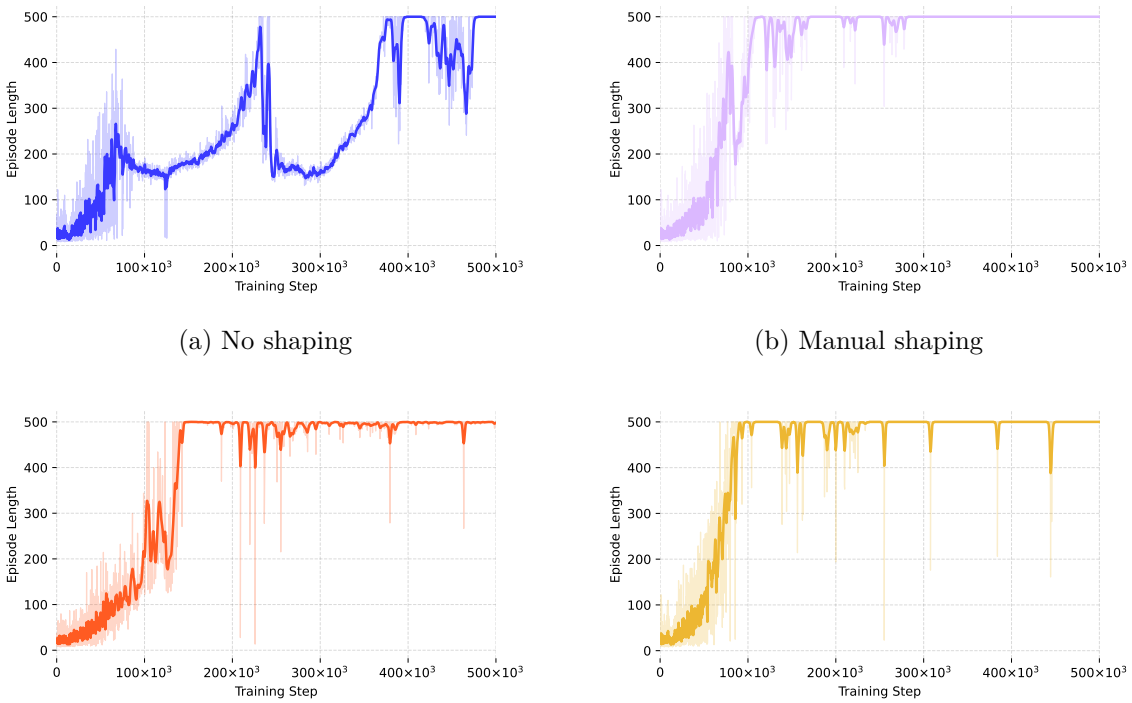
`sub(1, div(add(abs(div(theta, 0.2095)), abs(div(x, 2.4))), 2))`.

in the GP representation. This reward function encourages the agent to keep the pole angle θ close to zero and the cart position x near the center. It penalizes deviations from these optimal values, normalized by their respective limits.

In the PPO setting, it can be observed that although randomly generated reward shaping accelerates convergence, agents still exhibit catastrophic forgetting even after reaching the maximum episode reward for extended periods. In contrast, evolution is able to discover functions that mitigate this behavior. For DQN, however, this pattern is less pronounced.

6.3.1 PPO Results

We evaluate the impact of different reward shaping strategies on PPO performance in the CartPole environment, as shown in Figure 6.7. These experiments aim to isolate the contri-



(a) No shaping

(b) Manual shaping

(c) Random shaping

(d) Evolved shaping

Figure 6.8: **Comparison of different reward shaping approaches in the DQN (off-policy) agent setting**. Episode length is used as the evaluation metric. Graph (a) depicts a setting with no shaping reward, (b) shows training with a manually-designed reward function, (c) uses a randomly generated function, and (d) uses a reward function evolved through genetic programming.

bution of shaping by comparing unshaped, manually shaped, randomly shaped, and evolved reward functions. Even in a simple setting, the results reveal substantial differences in learning, especially in terms of speed and stability. The unshaped agent, depicted in Figure 6.7a struggles with high variance and slow convergence, while manual shaping in Figure 6.7b accelerates learning and consistency, demonstrating the benefit of expert-designed rewards.

Interestingly, randomly generated shaping functions also lead to fast early learning, as shown in Figure 6.7c. However, they introduce occassional instability as the training continues. This suggests that even unstructured signals can help guide exploration in low-complexity environments. However, their lack of reliability underscores the need for more principled approaches. The strongest results come from evolved shaping rewards, discovered via genetic programming. These consistently yield rapid convergence and stable performance, outperforming both manual and random alternatives. In Figure 6.7d, learning with evolved shaping converges within 20,000 steps, which is approximately **15x faster** than the no-shaping baseline. This highlights the potential of automated reward design, especially when domain knowledge is limited, or unavailable.

Fututre experiments in more complex environments are needed to further validate these finidings and better distinguish the value of structured, evolved shaping over random search.

6.3.2 DQN Results

In this section, we evaluate the impact of different reward shaping strategies on DQN performance, as shown in Figure 6.8. Unlike PPO, the results indicate that the three shaping approaches, manual (Figure 6.8b), random (Figure 6.8c), and evolved (Figure 6.8d) yield a very similar improvement in learning dynamics. Each shaped agent quickly converges to near-optimal performance with stable episode lengths, while the unshaped agent learns more slowly and with higher variance in episode lengths. The minimal differences between shaping methods suggest that, for DQN in this relatively simple environment, any additional shaping signal, whether structured or not, is sufficient to effectively guide learning. This robustness might stem from DQN's off-policy nature and experience replay, which help stabilize training even with noisy or imperfect reward signals.

These findings imply that the benefit of sophisticated, evolved shaping functions may be limited for off-policy methods like DQN in low-complexity tasks. To better assess the advantages of evolutionary reward design, future work should explore more challenging environments where baseline performance is weaker, and reward design plays a more critical role. The setting of evolution might also benefit from further exploration of alternative fitness functions.

6.4 Experiment Dimension C: Performance in the Transfer Learning Scenario

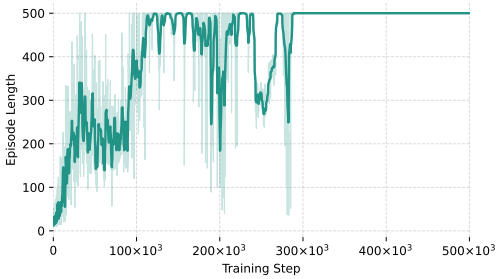
This experiment investigates the ability of a PPO agent to adapt to variations in environment dynamics when trained from scratch in each setting. Specifically, we evaluate whether a reward shaping function evolved in the original environment can effectively generalize to modified environments with altered physical parameters, details of which are introduced in Section 5.1.3. By comparing learning curves with and without the evolved reward shaping, we aim to determine whether the evolved signal can consistently accelerate the learning and improve stability across diverse dynamics, thereby facilitating transfer learning without additional hyperparameter tuning. The learning curves in settings without and with shaping are shown in Figure 6.9 and Figure 6.10, respectively.

The results demonstrate that even small modifications to the environment parameters (e.g., pole length, cart mass, gravity) can severely impair the agent's learning ability when no reward shaping is applied. This instability manifests itself as prolonged training plateaus, performance regressions, and high variance across runs, leading to the need for extensive fine-tuning for each environment variation. In contrast, the evolved reward shaping shows promise in mitigating these challenges by providing consistent and informative feedback that generalizes well across dynamics.

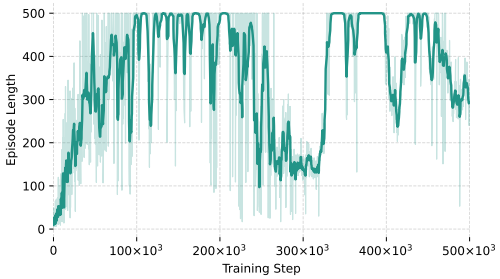
6.4.1 Transfer with No Shaping

The agent's unshaped learning curves presented in Figure 6.9 reveal three key limitations:

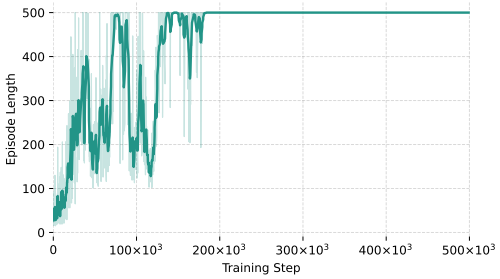
- **Sensitivity to dynamics changes:** Minor parameter adjustments disrupt learning trajectories, often requiring complete re-tuning.
- **Exploration inefficiency:** Sparse rewards lead to erratic policy updates, as seen in the high variance in episode length during the initial training phase.



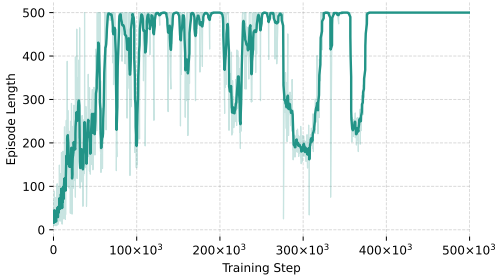
(a) Base CartPole



(b) ShortPole

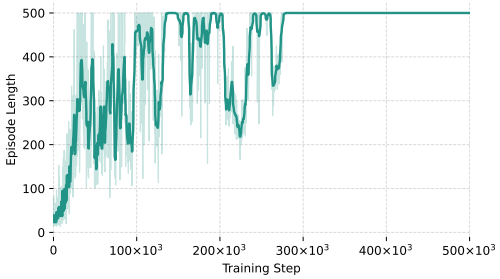


(c) LongPole

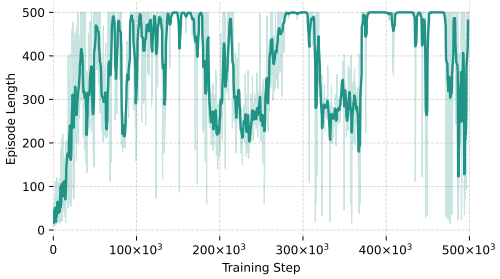


(d) HeavyPole

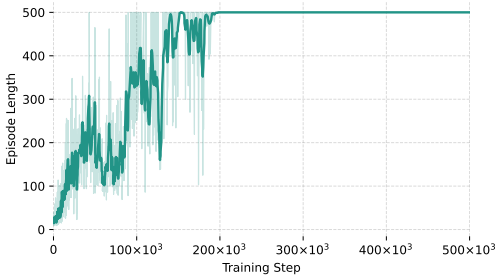
(e) LightPole



(f) HeavyCart

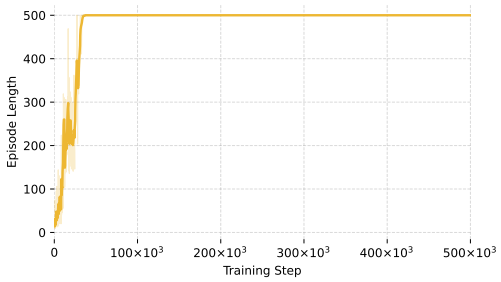


(g) LowGravity

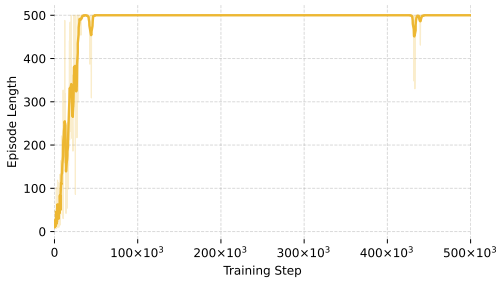


(h) HighGravity

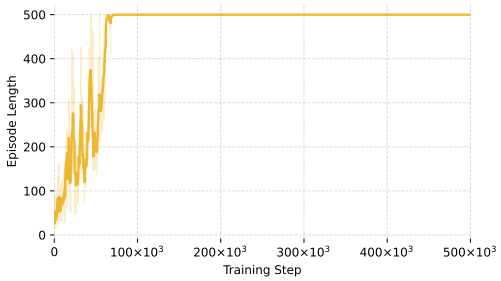
Figure 6.9: **Episode length over time in the transfer learning setup using PPO without reward shaping.** The plots display results across eight environment variations, each differing in dynamics parameters such as pole length and cart mass. The y-axis indicates episode length, while the x-axis shows training steps. This visualization highlights the agent's adaptability and performance across altered environments without additional shaping signals.



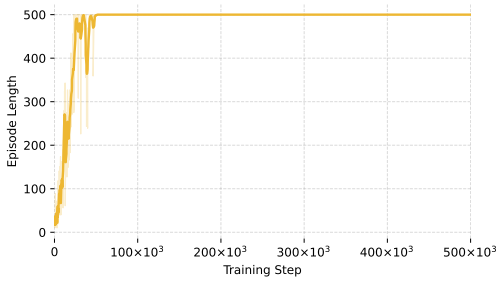
(a) Base CartPole



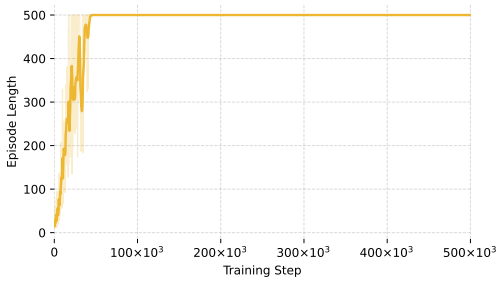
(b) ShortPole



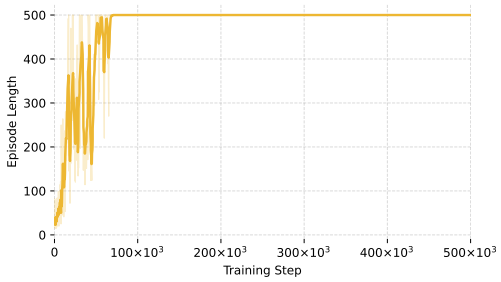
(c) LongPole



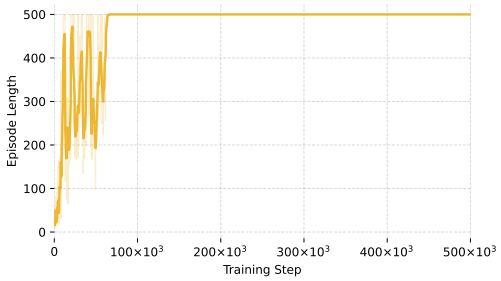
(d) HeavyPole



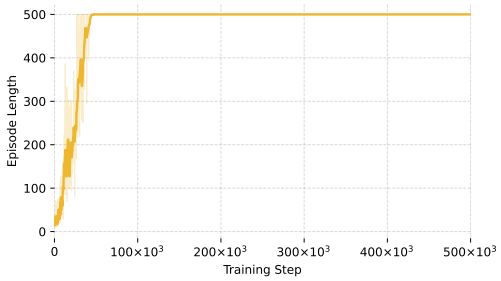
(e) LightPole



(f) HeavyCart



(g) LowGravity



(h) HighGravity

Figure 6.10: **Episode length over time in the transfer learning setup using PPO with evolved reward shaping functions.** The plots display results across eight environment variations, each differing in dynamics parameters such as pole length and cart mass. The y-axis indicates episode length, while the x-axis shows training steps. This visualization highlights the agent's adaptability and performance across altered environments without additional shaping signals. Evolved expression `mul(theta, sub(sub(theta, omega), omega))` was used for shaping.

- **Catastrophic forgetting:** Performance drops in later training stages suggest interference between new experiences and previously learned behaviors.

These issues align with the known brittleness of RL agents in sparse reward settings, where they struggle to correlate actions with delayed outcomes without explicit guidance.

6.4.2 Transfer with Evolved Shaping

The evolved shaping function addresses these limitations through:

- **Generalized reward signal:** By automatically encoding task-invariant features such as pole angle or cart velocity bounds, the shaping function provides consistent gradients even under perturbed dynamics.
- **Implicit curriculum:** The evolved rewards structure exploration, gradually focusing the agent on higher-precision adjustments as training progresses.
- **Reduced hyperparameter dependence:** Stable convergence across all eight environments suggests that the shaping function compensates for suboptimal setting or RL agent's hyperparameters.

Although traditional agent implementation often requires per-environment fine-tuning, evolved shaping decouples performance from specific dynamics parameters. This is specifically evident in the `HighGravity` (Figure 6.9h and Figure 6.10h), and `LightPole` environments (Figure 6.9e, and Figure 6.10e), where the shaped agents **achieve optimal policies** 20–30× **faster** than their unshaped counterparts. Similarly, results on `ShortPole` (Figure 6.9b and Figure 6.10b), and `LowGravity` (Figure 6.9g and Figure 6.10g) show that the unshaped version fails to learn the task altogether. The results suggest that this approach to shaping can partially offset the RL agent's sensitivity to environment and hyperparameter variations.

6.5 Summary of Experiments

Across all experiments, we observe that carefully tuned evolutionary setups, particularly those favoring larger populations and balanced selection pressure, yield more consistent and higher-quality shaping functions. These results guided the configuration used in the main reward evolution experiments. Furthermore, the comparative analysis of shaping strategies shows that evolved rewards not only accelerate learning, but also help mitigate catastrophic forgetting in on-policy settings like PPO, outperforming both manual and randomly generated alternatives. While off-policy methods like DQN appear less sensitive to the structure of shaping in simple environments, further experiments in more complex or non-stationary tasks are needed to fully assess the value of evolved shaping. In general, the findings of this chapter support the promise of genetic programming as a viable approach to automated reward design.

Chapter 7

Conclusion

This thesis explored the integration of evolutionary algorithms into reinforcement learning, with the aim of improving training efficiency and stability. Specifically, the focus was put on reward shaping, a technique that introduces intermediate rewards into the environment to provide the agent with more informative feedback and enhance the agent-environment interaction loop. To achieve this, genetic programming was employed to evolve tailored mathematical functions that serve as shaping functions. The experimental results show that the evolutionary approach can significantly accelerate learning and its convergence. In some cases, a well-evolved reward function enabled the agent to converge up to 15× faster compared to the baseline without reward shaping. Furthermore, evolved shaping functions were found to effectively mitigate issues such as catastrophic forgetting and reoccurring performance degradation observed with randomly generated rewards. While the random reward shaping approach also provided some improvements over the baseline, the evolved functions outperformed it in on-policy settings, suggesting that further training in more complex environments might be needed to widen the performance gap of these two approaches. Additional experiments indicated that certain evolved functions enabled agents to generalize across modified versions of the target environment, eliminating the need for fine-tuning the agent itself. This highlights one of the key advantages of the proposed method: although evolutionary computation incurs computational overhead, it shifts this burden away from the additional cost of fine-tuning in new task settings, as well as from the reliance on deep domain-specific knowledge required in the manual design scenario.

Compared to the state-of-the-art, this thesis extends existing work on evolutionary reward shaping by applying genetic programming not only to accelerate learning but also to address the challenge of catastrophic forgetting, an area not directly targeted in prior research. While earlier approaches have evolved reward functions for transferability, interpretability, or exploration, this work introduces forgetting-aware fitness criteria that explicitly promote generalization and stability across tasks. In doing so, it builds on and complements frameworks such as PushGP [70] and AutoRL [19], while contributing novel insights into the design of robust reward functions and transfer learning.

These promising initial results suggest substantial potential for future work. Follow-up research could extend this approach to more complex environments, such as MuJoCo, and explore different learning scenarios including task switching and continual learning. This setup might benefit from evolving ensembles of reward functions using a vectorized approach that could offer further improvements in performance and robustness. Additionally, investigating this method within the framework of potential-based reward shaping may also be crucial for certain applications. The broader impact of this work lies in its potential to

reduce reward hacking, a critical challenge in deploying safe and reliable reinforcement learning systems. Establishing a tighter and more controlled means of communication between the agent and the environment has the potential to alleviate the problems of misalignment stemming from improper reward design. By automating and improving the process of reward design, this work contributes to advancing the use of RL in real-world applications such as autonomous systems, robotics, and AI-driven decision-making.

Bibliography

[1] *MuJoCo: Multi-Joint dynamics with Contact* https://mujoco.org/. 2022. Available at: https://github.com/deepmind/mujoco. Physics engine and simulator developed by DeepMind, open-source since 2022.

[2] Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R. et al. Hindsight experience replay. *Advances in neural information processing systems*, 2017, vol. 30.

[3] Åström, K. J. Optimal control of Markov processes with incomplete state information I. *Journal of mathematical analysis and applications*. Elsevier, 1965, vol. 10, p. 174–205.

[4] Babaeizadeh, M.; Frosio, I.; Tyree, S.; Clemons, J. and Kautz, J. Reinforcement learning through asynchronous advantage actor-critic on a gpu. *ArXiv preprint arXiv:1611.06256*, 2016.

[5] Badnava, B.; Esmaeili, M.; Mozayani, N. and Zarkesh Ha, P. A new potential-based reward shaping for reinforcement learning agent. In: IEEE. *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*. 2023, p. 01–06.

[6] Bai, H.; Cheng, R. and Jin, Y. Evolutionary reinforcement learning: A survey. *Intelligent Computing*. AAAS, 2023, vol. 2, p. 0025.

[7] Banzhaf, W.; Machado, P. and Zhang, M. *Handbook of Evolutionary Machine Learning*. Springer, 2023.

[8] Barth Maron, G.; Hoffman, M. W.; Budden, D.; Dabney, W.; Horgan, D. et al. Distributed distributional deterministic policy gradients. *ArXiv preprint arXiv:1804.08617*, 2018.

[9] Bellman, R. A Markovian decision process. *Journal of mathematics and mechanics*. JSTOR, 1957, p. 679–684.

[10] Bellman, R. Dynamic programming. *Science*. American Association for the Advancement of Science, 1966, vol. 153, no. 3731, p. 34–37.

[11] Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research*, 2012, vol. 13, no. 2.

[12] Bertini, I.; De Felice, M.; Moretti, F. and Pizzuti, S. Start-up optimisation of a combined cycle power plant with multiobjective evolutionary algorithms. In:

Springer. *European Conference on the Applications of Evolutionary Computation*. 2010, p. 151–160.

[13] Bertsekas, D. *Dynamic programming and optimal control: Volume I*. Athena scientific, 2012.

[14] Bertsekas, D. P. Pathologies of temporal difference methods in approximate dynamic programming. In: IEEE. *49th IEEE Conference on Decision and Control (CDC)*. 2010, p. 3034–3039.

[15] Bodnar, C.; Day, B. and Lió, P. Proximal distilled evolutionary reinforcement learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020, vol. 34, no. 04, p. 3283–3290.

[16] Cao, X.-R. Stochastic learning and optimization-a sensitivity-based approach. *IFAC Proceedings Volumes*. Elsevier, 2008, vol. 41, no. 2, p. 3480–3492.

[17] Chaslot, G.; Bakkes, S.; Szita, I. and Spronck, P. Monte-carlo tree search: A new framework for game ai. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2008, vol. 4, no. 1, p. 216–217.

[18] Cheng, C.-A.; Kolobov, A. and Swaminathan, A. *Heuristic-Guided Reinforcement Learning*. 2021. Available at: https://arxiv.org/abs/2106.02757.

[19] Chiang, H.-T. L.; Faust, A.; Fiser, M. and Francis, A. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*. IEEE, 2019, vol. 4, no. 2, p. 2007–2014.

[20] Dayan, P. and Watkins, C. Q-learning. *Machine learning*, 1992, vol. 8, no. 3, p. 279–292.

[21] DeepMind. *Specification Gaming: The Flip Side of AI Ingenuity*. 2020. Available at: https://deepmind.com/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity. Accessed: 2025-01-21.

[22] Ding, Z. and Dong, H. Challenges of reinforcement learning. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Springer, 2020, p. 249–272.

[23] Eiben, A. E. Multi-parent recombination. *Evolutionary computation*, 1997, vol. 1, p. 289–307.

[24] Eiben, A. E. Multiparent recombination in evolutionary computing. In: *Advances in evolutionary computing: theory and applications*. Springer, 2003, p. 175–192.

[25] Eiben, A. E. and Smith, J. E. *Introduction to evolutionary computing*. Springer, 2015.

[26] Elfwing, S.; Uchibe, E.; Doya, K. and Christensen, H. I. Co-evolution of shaping rewards and meta-parameters in reinforcement learning. *Adaptive Behavior*. SAGE Publications Sage UK: London, England, 2008, vol. 16, no. 6, p. 400–412.

[27] Eriksson, A.; Capi, G. and Doya, K. Evolution of meta-parameters in reinforcement learning algorithm. In: IEEE. *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*. 2003, vol. 1, p. 412–417.

[28] Fernandez, F. C. and Caarls, W. Parameters tuning and optimization for reinforcement learning algorithms using evolutionary computing. In: IEEE. *2018 International Conference on Information Systems and Computer Science (INCISCOS)*. 2018, p. 301–305.

[29] Fisher, D. G. Process control: an overview and personal perspective. *The Canadian Journal of Chemical Engineering*. Wiley Online Library, 1991, vol. 69, no. 1, p. 5–26.

[30] Fortin, F.-A.; De Rainville, F.-M.; Gardner, M.-A.; Parizeau, M. and Gagné, C. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*. JMLR. org, 2012, vol. 13, Jul, p. 2171–2175.

[31] Foundation, F. *Gymnasium: CartPole Environment* https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/classic_control/cartpole.py. 2023. Accessed: 2025-05-06.

[32] Foundation, T. F. *Gymnasium: A Standard API for Reinforcement Learning Environments* https://gymnasium.farama.org/. Farama Foundation, 2023.

[33] Franke, J. K.; Köhler, G.; Biedenkapp, A. and Hutter, F. Sample-efficient automated deep reinforcement learning. *ArXiv preprint arXiv:2009.01555*, 2020.

[34] French, R. M. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*. Elsevier, 1999, vol. 3, no. 4, p. 128–135.

[35] Gaier, A. and Ha, D. Weight agnostic neural networks. *Advances in neural information processing systems*, 2019, vol. 32.

[36] Gosavi, A. et al. *Simulation-based optimization*. Springer, 2015.

[37] Gravina, D.; Liapis, A. and Yannakakis, G. Surprise search: Beyond objectives and novelty. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 2016, p. 677–684.

[38] Grześ, M. Reward Shaping in Episodic Reinforcement Learning. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2017, p. 565–573. AAMAS '17.

[39] Gupta, A.; Mendonca, R.; Liu, Y.; Abbeel, P. and Levine, S. Meta-reinforcement learning of structured exploration strategies. *Advances in neural information processing systems*, 2018, vol. 31.

[40] Gupta, A.; Pacchiano, A.; Zhai, Y.; Kakade, S. M. and Levine, S. *Unpacking Reward Shaping: Understanding the Benefits of Reward Engineering on Sample Complexity*. 2022. Available at: https://arxiv.org/abs/2210.09579.

[41] Hester, T.; Vecerik, M.; Pietquin, O.; Lanctot, M.; Schaul, T. et al. Deep q-learning from demonstrations. In: *Proceedings of the AAAI conference on artificial intelligence*. 2018, vol. 32, no. 1.

[42] Holland, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[43] Huang, S.; Dossa, R. F. J.; Raffin, A.; Kanervisto, A. and Wang, W. The 37 Implementation Details of Proximal Policy Optimization. In: *ICLR Blog Track*. 2022. Available at: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. Accessed: 2025-05-04.

[44] Huang, Y. *CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms* https://github.com/vwxyzjn/cleanrl. 2022. Accessed: 2025-05-04.

[45] Hussein, A.; Gaber, M. M.; Elyan, E. and Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*. ACM New York, NY, USA, 2017, vol. 50, no. 2, p. 1–35.

[46] Isele, D. and Cosgun, A. Selective experience replay for lifelong learning. In: *Proceedings of the AAAI conference on artificial intelligence*. 2018, vol. 32, no. 1.

[47] Jaderberg, M.; Dalibard, V.; Osindero, S.; Czarnecki, W. M.; Donahue, J. et al. Population based training of neural networks. *ArXiv preprint arXiv:1711.09846*, 2017.

[48] Khadka, S. and Tumer, K. Evolutionary reinforcement learning. *ArXiv preprint arXiv:1805.07917*, 2018, vol. 223.

[49] Khurma, R. A.; Aljarah, I.; Sharieh, A. and Mirjalili, S. Evolopy-fs: An open-source nature-inspired optimization framework in python for feature selection. *Evolutionary machine learning techniques: Algorithms and applications*. Springer, 2020, p. 131–173.

[50] Kirkpatrick, J.; Pascanu, R.; Rabinowitz, N.; Veness, J.; Desjardins, G. et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*. Proceedings of the National Academy of Sciences, march 2017, vol. 114, no. 13, p. 3521–3526. ISSN 1091-6490. Available at: http://dx.doi.org/10.1073/pnas.1611835114.

[51] Koza, J. R. Genetic programming: on the programming of computers by means of natural selection Cambridge. *MA: MIT Press.[Google Scholar]*, 1992.

[52] Koza, J. On the programming of computers by means of natural selection. *Genetic programming*. MIT press, 1992.

[53] Lampinen, J. and Zelinka, I. Mechanical engineering design optimization by differential evolution. In: *New ideas in optimization*. 1999, p. 127–146.

[54] Larkin, F. and Ryan, C. Modesty is the best policy: Automatic discovery of viable forecasting goals in financial data. In: Springer. *European Conference on the Applications of Evolutionary Computation*. 2010, p. 202–211.

[55] Laud, A. and DeJong, G. The influence of reward on the speed of reinforcement learning: An analysis of shaping. In: *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 2003, p. 440–447.

[56] Laud, A. D. *Theory and application of reward shaping in reinforcement learning*. University of Illinois at Urbana-Champaign, 2004.

[57] Lehman, J. and Stanley, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*. MIT Press, 2011, vol. 19, no. 2, p. 189–223.

[58] Liu, Y.; Sun, Y.; Xue, B.; Zhang, M.; Yen, G. G. et al. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*. IEEE, 2021, vol. 34, no. 2, p. 550–570.

[59] Marchesini, E.; Corsi, D. and Farinelli, A. Genetic soft updates for policy evolution in deep reinforcement learning. In: *International Conference on Learning Representations*. 2020.

[60] Marchesini, E.; Corsi, D. and Farinelli, A. Exploring safer behaviors for deep reinforcement learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2022, vol. 36, no. 7, p. 7701–7709.

[61] McDougall, C. *The ARENA Tutorial Series* https://github.com/callummcdougall/arena. 2022. Accessed: 2025-05-04.

[62] Mnih, V. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv preprint arXiv:1602.01783*, 2016.

[63] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I. et al. Playing atari with deep reinforcement learning. *ArXiv preprint arXiv:1312.5602*, 2013.

[64] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J. et al. Human-level control through deep reinforcement learning. *Nature*. Nature Publishing Group, 2015, vol. 518, no. 7540, p. 529–533.

[65] Moriarty, D. E.; Schultz, A. C. and Grefenstette, J. J. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research*, 1999, vol. 11, p. 241–276.

[66] Nair, A.; Srinivasan, P.; Blackwell, S.; Alcicek, C.; Fearon, R. et al. Massively parallel methods for deep reinforcement learning. *ArXiv preprint arXiv:1507.04296*, 2015.

[67] Nanda, N.; Chan, L.; Lieberum, T.; Smith, J. and Steinhardt, J. Progress measures for grokking via mechanistic interpretability. *ArXiv preprint arXiv:2301.05217*, 2023.

[68] Ng, A. Y.; Harada, D. and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In: *Icml*. 1999, vol. 99, p. 278–287.

[69] Ng, A. Y.; Harada, D. and Russell, S. J. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 278–287. ICML '99. ISBN 1558606122.

[70] Niekum, S.; Barto, A. G. and Spector, L. Genetic programming for reward function search. *IEEE Transactions on Autonomous Mental Development*. IEEE, 2010, vol. 2, no. 2, p. 83–90.

[71] Nordin, P.; Keller, R. and Francone, F. Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications. In:. January 1998.

[72] Olds, J. and Milner, P. Positive reinforcement produced by electrical stimulation of septal area and other regions of rat brain. *Journal of comparative and physiological psychology*. American Psychological Association, 1954, vol. 47, no. 6, p. 419.

[73] OpenAI. *OpenAI Five defeats Dota 2 world champions* https://openai.com/index/openai-five-defeats-dota-2-world-champions/. 2019. Accessed: 2025-01-18.

[74] Powell, W. B. *Approximate Dynamic Programming: Solving the curses of dimensionality*. John Wiley & Sons, 2007.

[75] Pugh, J. K.; Soros, L. B. and Stanley, K. O. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*. Frontiers, 2016, vol. 3, p. 202845.

[76] Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A. and Dormann, N. *RL Baselines3 Zoo: A collection of pre-trained Reinforcement Learning agents and RL training framework* https://github.com/DLR-RM/rl-baselines3-zoo. 2021. Accessed: 2025-05-04.

[77] Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M. et al. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 2021. Software available at https://github.com/DLR-RM/stable-baselines3.

[78] Raffin, A.; Hill, A.; Kanervisto, A.; Ernestus, M. and Dormann, N. *Stable-Baselines3: Reliable Reinforcement Learning Implementations* https://github.com/DLR-RM/stable-baselines3. 2021. Accessed: 2025-05-04.

[79] Rahmat Samii, Y. and Michielssen, E. Electromagnetic optimization by genetic algorithms. *Microwave Journal*. Horizon House Publications, Inc., 1999, vol. 42, no. 11, p. 232–232.

[80] Ryan, R. M. and Deci, E. L. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology*. Elsevier, 2000, vol. 25, no. 1, p. 54–67.

[81] Salimans, T.; Ho, J.; Chen, X.; Sidor, S. and Sutskever, I. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. Available at: https://arxiv.org/abs/1703.03864.

[82] Schmidt, M.; Safarani, S.; Gastinger, J.; Jacobs, T.; Nicolas, S. et al. On the performance of differential evolution for hyperparameter tuning. In: IEEE. *2019 international joint conference on neural networks (IJCNN)*. 2019, p. 1–8.

[83] Schulman, J.; Moritz, P.; Levine, S.; Jordan, M. and Abbeel, P. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. Available at: https://arxiv.org/abs/1506.02438.

[84] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A. and Klimov, O. Proximal policy optimization algorithms. *ArXiv preprint arXiv:1707.06347*, 2017.

[85] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A. and Klimov, O. *Proximal Policy Optimization Algorithms*. 2017. Available at: https://arxiv.org/abs/1707.06347.

[86] Schultz, W. *Reward signals. Scholarpedia 2: 2184*. 2007.

[87] Schultz, W. Reward. *Scholarpedia*, 2007, vol. 2, no. 3, p. 1652.

[88] Silver, D. *Lectures on Reinforcement Learning* URL: https://www.davidsilver.uk/teaching/. 2015.

[89] Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L. et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, january 2016, vol. 529, p. 484–489.

[90] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M. et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. American Association for the Advancement of Science, december 2018, vol. 362, no. 6419, p. 1140–1144.

[91] Singh, S.; Lewis, R. L.; Barto, A. G. and Sorg, J. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*. IEEE, 2010, vol. 2, no. 2, p. 70–82.

[92] Skalse, J.; Howe, N.; Krasheninnikov, D. and Krueger, D. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 2022, vol. 35, p. 9460–9471.

[93] Skinner, B. F. *The behavior of organisms: An experimental analysis*. BF Skinner Foundation, 2019.

[94] Stanley, K. O.; Clune, J.; Lehman, J. and Miikkulainen, R. Designing neural networks through neuroevolution. *Nature Machine Intelligence*. Nature Publishing Group UK London, 2019, vol. 1, no. 1, p. 24–35.

[95] Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*. MIT Press, 2002, vol. 10, no. 2, p. 99–127.

[96] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. Secondth ed. The MIT Press, 2018. Available at: http://incompleteideas.net/book/the-book-2nd.html.

[97] Szepesvári, C. *Algorithms for Reinforcement Learning*. Morgan and Claypool, July 2010. Available at: http://www.ualberta.ca/~szepesva/RLBook.html.

[98] Tsitsiklis, J. N. and Roy, B. V. Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives. *IEEE Trans. Autom. Control.*, 1999, vol. 44, p. 1840–1851. Available at: https://api.semanticscholar.org/CorpusID:15656556.

[99] Van Hasselt, H.; Doron, Y.; Strub, F.; Hessel, M.; Sonnerat, N. et al. Deep reinforcement learning and the deadly triad. *ArXiv preprint arXiv:1812.02648*, 2018.

[100] Vecerik, M.; Hester, T.; Scholz, J.; Wang, F.; Pietquin, O. et al. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *ArXiv preprint arXiv:1707.08817*, 2017.

[101] Watkins, C. J. and Dayan, P. Q-learning. *Machine learning*. Springer, 1992, vol. 8, p. 279–292.

[102] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*. Springer, 1992, vol. 8, p. 229–256.

[103] Yu, Y. Towards Sample Efficient Reinforcement Learning. In: *IJCAI*. 2018, p. 5739–5743.

[104] Zhang, A.; Ballas, N. and Pineau, J. A dissection of overfitting and generalization in continuous reinforcement learning. *ArXiv preprint arXiv:1806.07937*, 2018.

Appendix A

Fixed Hyperparameters of RL Agents

This appendix contains the fixed hyperparameters used in the implementations of RL algorithms. These were adapted from state-of-the-art implementations to ensure the learnability of the task under the given settings. For reproducibility, a fixed random seed was used in all experiments. The seed for each run is set as:

$$\texttt{seed} = 42 + \texttt{run\_id}$$

where `run_id` is an index of a particular run. Unless stated otherwise, a particular configuration of the experiment was run multiple times, ensuring variation between runs while allowing for reproducibility of each experiment.

A.1 DQN Parameters

```
env_id: str = "CartPole-v1"
num_envs: int = 1

# Duration of different phases / buffer memory settings
total_timesteps: int = 500_000
steps_per_train: int = 10
trains_per_target_update: int = 100
buffer_size: int = 10_000

# Optimization hparams
batch_size: int = 128
learning_rate: float = 2.5e-4

# RL-specific
gamma: float = 0.99
exploration_fraction: float = 0.2
start_e: float = 1.0
end_e: float = 0.1

total_timesteps - self.buffer_size >= self.steps_per_train
```

```
total_training_steps =
    (self.total_timesteps - self.buffer_size) // self.steps_per_train
```

A.2 PPO Parameters

```
env_id: str = "CartPole-v1"
mode: Literal["classic-control", "atari", "mujoco"] = "classic-control"

# Duration of different phases
total_timesteps: int = 500_000
num_envs: int = 4
num_steps_per_rollout: int = 128
num_minibatches: int = 4
batches_per_learning_phase: int = 4

# Optimization hyperparameters
lr: float = 2.5e-4
max_grad_norm: float = 0.5

# RL hyperparameters
gamma: float = 0.99

# PPO-specific hyperparameters
gae_lambda: float = 0.95
clip_coef: float = 0.2
ent_coef: float = 0.01
vf_coef: float = 0.25

batch_size = self.num_steps_per_rollout * self.num_envs
minibatch_size = self.batch_size // self.num_minibatches
total_phases = self.total_timesteps // self.batch_size
total_training_steps =
    self.total_phases * self.batches_per_learning_phase * self.num_minibatches
```

Appendix B

Content of the Associated Cloud Storage

`dip/src/`	Folder containing code for the implementation of experiments.
`dip/results/`	Folder containing subset of results plotted in the thesis.
`dip/README.md`	Description of the code and tutorial on how to run it.
`dip/requirements.txt`	List of required Python libraries.
`src-tech-report/`	Folder with LaTeX source files.
`xgulci00-thesis.pdf`	Electronically submitted version of the thesis.
`xgulci00-thesis-print.pdf`	Printed version of the thesis[1].