UCLouvain

École polytechnique de Louvain

Combining Evolutionary and Curiosity-Driven Algorithms to Enhance and Adapt Exploration Efficiency in Q-Learning

Authors: **Lucas Suetens, Quentin Boulanger**
Supervisor: **Siegfried Nijssen**
Readers: **Eric Piette, Achille Morenville**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

Reinforcement Learning (RL) performance is highly sensitive to the hyper-parameter values used during training. However, tuning these parameters often relies on experimental methods guided by intuition or computationally expensive techniques such as grid search and random search, which can lead to suboptimal performance. To address this issue, Automated Reinforcement Learning (AutoRL) has gained significant attention in recent years.

This master's thesis explores the development of a novel exploration strategy that combines AutoRL with a curiosity-driven approach, aiming to generalize the well-known challenge of the exploration-exploitation trade-off. Our approach integrates the Never Give Up (NGU) method developed by Badia et al with online tuning mechanisms inspired by Evolutionary Approachs, and is applied to the widely-used Q-learning algorithm.

This work contributes to the field of Automated Reinforcement Learning by introducing a general exploration strategy that eliminates the need for hyper-parameter tuning, thereby enhancing the efficiency and applicability of reinforcement learning algorithms. Our approach surpasses simpler implementations of evolutionary algorithms applied to Q-learning. These advancements have the potential to make reinforcement learning more accessible to practitioners who are not experts in the field, thereby broadening the scope of its real-world applications.

Contents

Chapter 1

Introduction

1.1 Background & Motivation

While Reinforcement Learning (RL) performance has been proved to be highly sensitive to the hyper-parameters used during training [6, 1, 4], selecting the best hyper-parameters for any given case still heavily relies on experimental methods. These methods are often guided by intuition or straightforward approaches such as grid search and random search. These approaches are computationally expensive, often suboptimal, and error-prone [12]. It has even been shown that this methodology explains why many RL algorithms that show great results in initially researched domains fail to generalize beyond these domains [11].

Recently, the field of Automated Machine Learning (AutoML) [8, 5] has emerged, revolutionizing this process by employing meta-learning algorithms that autonomously optimize hyper-parameters in machine learning models. These advancements have recently begun to spread to Reinforcement Learning with the development of Automated Reinforcement Learning (AutoRL) [12]. AutoRL requires specific algorithms because of the inherent differences between ML and RL hyper-parameters and their training environments [12] .

Additionally, a well-known dilemma in the training of all reinforcement learning models is the Exploration-Exploitation Trade-off [9]. This dilemma involves the

decision the algorithm must make at each training step on whether to explore entirely new states, thereby expanding its knowledge base, or to exploit its existing knowledge by concentrating training on the most promising states. This trade-off is also referred to as the Exploration Challenge.

To address this dilemma, an innovative solution was developed by Badia et al. (2020) [2]. They introduced a curiosity-driven approach that incentivizes exploration. This method has been widely recognized by the Reinforcement Learning community, as it led to Agent57 [13], the first agent to outperform humans on all Atari games. However, the performance of this method still relies on per-environment hyper-parameter fine-tuning.

In this master's thesis, we aim to develop a new exploration strategy that can adapt to any given environment without the need for fine-tuning. This approach will integrate online hyper-parameter tuning based on Automated Reinforcement Learning (AutoRL) with the curiosity-driven method developed by Badia et al. [2]. A major contribution of this thesis will be to adapt these algorithms, originally designed for Deep Q-Networks (DQN), to basic Q-learning, a simple but effective framework developed by Watkins and Dayan in 1992 [17].

By doing so, we will explore the application of AutoRL to the critical challenge of the Exploration-Exploitation Trade-off.

1.2 Structure of the thesis

In Chapter 2, we will begin the thesis by defining key concepts and mathematical notations essential for understanding the problem we aim to solve and the notations used throughout the paper.

In Chapter 3, we will provide the necessary background to fully grasp our methodology and the key concepts underlying our research. This includes a detailed explanation of the specific algorithms we employ, ensuring a comprehensive understanding of our approach.

In Chapter 4, we will describe our implementation, discussing the rationale behind our choices and the adaptations made to apply these methods to Q-learning.

In Chapters 5 and 6, we will present the results of our algorithms across various environments. To generalize our findings, we will test our algorithms in four distinct settings and analyze the insights they provide regarding the exploration-exploitation trade-off.

Finally, in Chapters 7 and 8, we will outline the limitations of our results and provide the conclusions of our research.

1.3 Implementation Code and GitHub Repository

The complete code for our implementation is available in a public GitHub repository. You can access it via the following link: https://github.com/QBoulanger/master-thesis-boulanger-suetens

Chapter 2

Problem Statement & Notations

2.1 Reinforcement learning

Reinforcement Learning (RL) is a subfield of machine learning where an agent learns to make decisions by performing actions in an environment to maximize some notion of cumulative reward [9]. Unlike supervised learning, where the model is trained on a fixed dataset, RL involves learning from the consequences of actions, typically through trial and error. An RL algorithm is generally characterized by the interaction between an agent and the environment, which can be described by a Markov Decision Process (see section 2.2 for more details).

At each time step (t), the agent (or actor) observes the current state s_t of the environment, selects an action a_t, and receives a reward r_t along with the next state s_{t+1}. These interactions are often grouped into episodes, which are sequences of states, actions, and rewards that start from an initial state and proceed until reaching a terminal state. The goal of the agent is to learn a policy π, which maps states to actions, that maximizes the expected sum of rewards over time, often referred to as the return.

Consider the classic game of Snake as an example. The agent (the snake) observes the current state s_t of the environment (the game board), which includes the positions of the snake and the food. At each time step, the snake selects an action

a_t (e.g., move left, right, up, or down), and the environment responds with a new state s_{t+1} and a reward r_t. In this context, the reward might be +1 for eating food, -1 for colliding with the wall or itself, and 0 for all other actions. An episode begins with a new game and ends when a termination condition is met, such as the snake winning by eating a certain amount of food, losing by colliding with a wall or itself, or reaching a maximum number of time steps. After each episode, the agent uses the knowledge gained from previous episodes to improve its performance in future games.

2.2 Markov Decision Process & generalisation of the AutoRL Problem

Every Reinforcement Learning environments studied in this research can be represented by a Markov Decision Process (MDP) defined as a 7-tuple $(S, A, P, R, \rho_0, T, \gamma)$, where S is the set of States, A is the set of Actions, $P(s_i, a, s_t) : S \times A \times S \to \mathbb{R}^+$ describes the transition dynamic (The probability that making the action a in state s_i results in state s_t), $R(s_i, a, s_t) : S \times A \times S \to \mathbb{R}$ describes the reward dynamic (if the agent makes the action a in state s_i, and results in state s_t what will be the reward), $\rho_0(s) : S \to \mathbb{R}^+$ is the initial state distribution (the probability that the system starts at state s), T is the set of terminal states and $\gamma : [0, 1]$ is the discount factor.

Even though we limited our study to MDP-representable environments, many of our results may be extendable to Partially-Observable Markov Decision Process (POMDP) where we add 2 additionnal components: $\mathcal{O}$ which is the set of Observations and $\Omega(s_1, a, s_2) : S \times A \times S \to [0, 1]$ which describes the probability density function that observation o occurs if making action a at state s. The resulting tuple is $(S, A, P, R, \mathcal{O}, \Omega, \rho_0, T, \gamma)$.

All reinforcement learning studied refer to the problem of finding a policy $\pi_\theta(s) : S \to A$ which is parameterized by θ that maximizes the cumulative reward

(J):

$$\max_{\theta} J(\theta, \zeta) \text{ where } J(\theta, \zeta) = \mathbb{E}_{\tau \sim \pi_\theta}[\sum_{t>=0} \gamma^t r_t] \tag{2.1}$$

where τ is the trajectory generated by the interaction between the policy π_θ and the environment and ζ is the set of hyperparameters studied by the AutoRL Algorithms (in our case the exploration-exploitation trade-off hyperparameters $\in \zeta$).

The AutoRL Problem goal is to find the best hyperparameters and so, the best values for ζ. In a general scenario we can define the AutoRL problem as a general bi-level optimization:

$$\max_{\zeta} f(\theta^*, \zeta) \text{ s.t. } \theta^* \in \arg\max_{\theta} J(\theta, \zeta) \tag{2.2}$$

Typically, we use the cumulative reward (J) as the outer-level evaluation objective: $f(\theta^*, \zeta) = J(\theta^*, \zeta)$. The AutoRL Problem can be defined as the bi-level optimization:

$$\max_{\zeta} J(\theta^*, \zeta) \text{ s.t. } \theta^* \in \arg\max_{\theta} J(\theta, \zeta) \tag{2.3}$$

2.3 Q-Learning Algorithms

In this research, we focused our attention on the Q-learning algorithm; an off-policy, model-free, value-based method developped in 1992 by Watkins and Dayan [17] that has showed promising results on various fields.

Q-learning seek to learn an action-value function (e.g. a function that predicts the expected discounted cumulative reward for taking any action a, on any state s). The optimal action-value function, also called optimal Q-function, and noted q_* can be described as follow:

$$q_*(s, a) = \max_{\pi} q_\pi(s, a) \text{ where } q_\pi(s, a) = \mathbb{E}_\pi[\sum_{k>=0} \gamma^k r_{t+k+1} | s_t = s, a_t = a] \tag{2.4}$$

$q_\pi(s, a)$ is the expected return from starting from state s at time t, taking action

a, and following policy π. For any state s and action a, the optimal action-value function ($q_*(s, a)$) gives the highest expected return achievable, starting with action a on state s, by following any policy π.

A fundamental property of q_* is that it follows the Bellman optimality equation:

$$q_*(s, a) = \mathbb{E}[r(s, a) + \gamma * \max_{a' \in A} q_*(s_{next}, a')] \tag{2.5}$$

where s_{next} is the resulting state after applying action a on state s, $r(s, a)$ is the actual reward for applying the action a on state s.

Q-learning uses this property to iteratively learn the action-value function. The algorithm uses a Q-table, a 2D array that stores the current Q-value for each action and state. This table is initialized with all Q-values set to 0. It then updates each value using the following algorithm:

$$Q(s, a) = Q(s, a) + \alpha * (r(s, a) + \gamma * \max_{a' \in A}(Q(s_{next}, a')) - Q(s, a)) \tag{2.6}$$

where $Q(s, a)$ is the Q-value for action a and state s, α is the learning rate, r(s, a) is the actual reward for applying the action a on state s, γ is the discount factor, $\max_{a' \in A}(Q(s_{next}, a'))$ is the highest expected reward for all actions applied to the next state (s_{next}) based on the current Q-table.

Watkins and Dayan demonstrated that the Q-learning algorithm (Algorithm 1) converges to the optimal Q-function under certain conditions, including an appropriate exploration policy and a sufficiently small learning rate [17].

Q-learning is an off-policy method because it learns the optimal policy independently of the agent's actions. It evaluates the potential future rewards from what is deemed the best action at each state, regardless of the agent's current policy or the action actually taken. This allows Q-learning to estimate the value of the optimal policy while the agent explores and learns from the environment using

Algorithm 1 Q-Learning Basic Algorithm

Initialize the value function $Q(s, a, \theta)$ with all Q-values set to 0
Initialize the learning rate α and the discount factor γ
while not converged **do**
    Get the initial state $s \in S$
    while s is not a terminal state **do**
        Select an action $a \in A$ using an ϵ-greedy policy (see Section 2.5) from Q
        Execute action a, receive reward r, and observe next state s'
        Update Q-value:

$$Q(s, a, \theta) \leftarrow Q(s, a, \theta) + \alpha \left[r + \gamma \max_{a' \in A} Q(s', a', \theta) - Q(s, a, \theta) \right]$$

        $s \leftarrow s'$
    end while
end while

a different, possibly exploratory policy. This characteristic distinguishes it from on-policy methods, where the learning is based on the evaluation of the actions dictated by the policy currently being followed by the agent.

Q-learning is a model-free method because it doesn't rely on a modelization of the problem. Instead, it learns from the experience of interacting with the environment, updating its policy based on the rewards received for actions taken in specific states. This approach allows the algorithm to learn the optimal policy for maximizing rewards without needing a predefined transition model of the environment. It opposes model-based methods that require a complete or partial knowledge of the environment's dynamics. Model-based approaches use this knowledge to simulate outcomes of possible actions, which can then be used to plan the best course of action ahead of time.

Q-learning is value-based because it focuses on learning the values of actions taken in particular states rather than directly determining the policy.

2.4 The Exploration-Exploitation Tradeoff

The exploration-exploitation trade-off [9] is a critical challenge in Reinforcement Learning. All Reinforcement learning models, including Q-learning, PPO, etc. have to choose at any given iteration step whether to explore entirely new states, even though seemingly-unoptimal, or whether to exploit the current knowledge of the database to concentrate training on the most promising states.

Let's imagine that at a given iteration step, and for a given state, the action-value function (Q) takes the following form (See Figure 2.1).

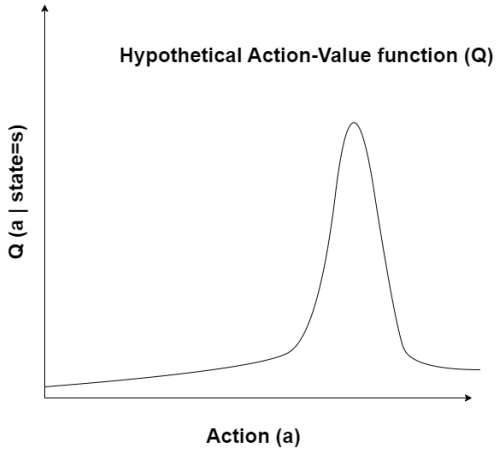


Figure 2.1: Hypothetical Action-Value function (Q)

Introducing the inverse temperature parameter, denoted as β, the action-selection probability (i.e., the probability that the algorithm selects a given action in this state) is defined as follows:

$$p(a|s) = \frac{e^{Q(s,a)\beta}}{\sum_{a' \in A} e^{Q(s,a')\beta}} \tag{2.7}$$

The parameter β controls the sharpness of the probability distribution, thereby parameterizing the balance between exploration and exploitation in the agent's learning process.

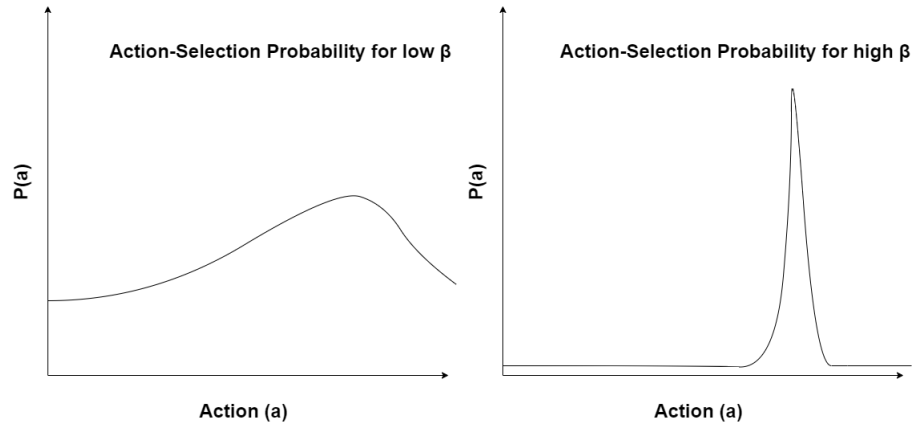


Figure 2.2: Comparison of Action-Selection Probability for Low and High Values of β

When β is low, we have high entropy in the Action-Selection Probability distribution and thus, the learning is characterized by high exploration and low exploitation. When β is high, we have low entropy in the Action-Selection Probability distribution and thus, the learning is characterized by low exploration and high exploitation.

If the algorithm excessively focus on exploitation (too high β), it may concentrate too much on a narrow set of state-action pairs. These pairs might seem to be the best choice according to the current action-value function (Q), but they might not lead to the optimal outcomes. This narrow focus can be inefficient because it might prevent the algorithm from considering better alternatives that could be more effective in the long run. And so, the learning of the optimal Q-function (Q^*) is hindered.

On the other hand, if the algorithm overemphasize exploration (too low β). This could make the algorithm's actions appear random and prevent it from sufficiently exploiting the knowledge of which actions have historically led to better rewards. As a result, this can also make the learning process of the optimal Q-function (Q^*) highly inefficient.

This is the reason why we observe that the choice of the β parameter has a tremendous impact on the model's performance, and the efficiency of the training.

Balancing between exploration and exploitation is essential for ensuring that the learning process is neither too greedy nor too random, and foster an effective approach to navigate and learn from the environment.

2.5 The Exploration-Exploitation Tradeoff in Q-learning

To parameterize the exploration-exploitation trade-off in Q-learning, we usually employ the Epsilon-Greedy Strategy.

This method hinges on a parameter, ϵ, which determines the likelihood of the algorithm choosing a random action from the set of possible actions (A) each time a decision is made. With probability ϵ, the algorithm explores by selecting a random action. On the other hand, with a probability of $1 - \epsilon$, the algorithm exploits its current knowledge by selecting the action with the highest Q-value.

$$a_t = \begin{cases} \text{random action from } A, & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a), & \text{with probability } 1 - \epsilon \end{cases} \tag{2.8}$$

It's also common in Q-learning to decay the Epsilon. Meaning that we usually starts with a high value for ϵ, allowing more exploration at the beginning, that we decay over time to become a lower value and thus turn to more exploitation. This decay can be linear or exponential.

Here is an example for exponential decay:

$$\epsilon = \epsilon_0 \cdot e^{-\text{decay rate} \times \text{episode number}} \tag{2.9}$$

where ϵ_0 is the initial epsilon value, the "decay rate" is a constant determining the speed of the decay, and "episode number" tracks the progression over time.

2.6 Sparse Reward vs Hard Exploration: Impact on the Exploration-Exploitation trade-off

The exploration-exploitation strategy that a model should use is highly dependent on the environment it is addressing. To categorize different environments, an interesting distinction is between Sparse Reward (Hard Exploration) and Dense Reward scenarios. We can observe that exploration-exploitation strategies that perform well in dense reward scenarios usually perform poorly in hard exploration scenarios, and vice versa. This notion has thus a significant impact on our objective to find a universal exploration-exploitation strategy that would perform well across all environment types.

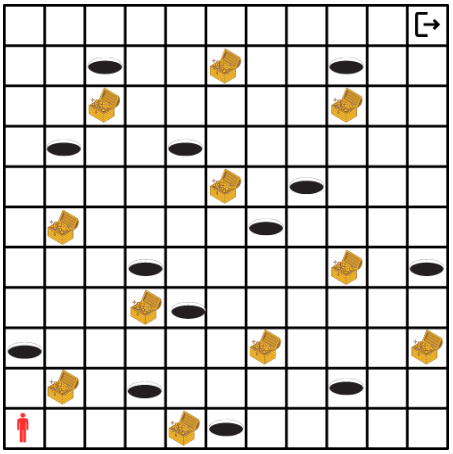


Figure 2.3: Example of a dense-reward scenario: Grid World (see Section 5.3 for more details)

In a dense reward scenario, the agent receives frequent feedback through rewards for its actions. This means that nearly every action taken by the agent provides some indication of its value, allowing the agent to learn more efficiently and adjust its strategy quickly. Dense reward environments typically make the learning process faster and more stable, as the agent can continuously refine its policy based on consistent feedback.

Figure 2.4: Example of a sparse-reward (hard-exploration) scenario: Frozen-Lake (see Section 5.4 for more details)

On the other hand, a sparse reward or hard exploration scenario is characterized by infrequent and delayed rewards. In these environments, the agent receives rewards only after completing a series of actions or reaching a long-term goal. This makes it difficult for the agent to understand which actions are beneficial, leading to challenges in learning effective policies. The agent must explore extensively to discover the sequence of actions that leads to a reward, which is computationally demanding and often leads to inefficient exploration.

2.7 Notation Table

Notation	Meaning
$\mathcal{S}$	State space
$\mathcal{A}$	Action space
P	Transition function
R	Reward function
$\mathcal{O}$	Observation space
ρ_0	Initial state distribution
T	Set of terminal states
γ	Discount factor
t	Inner loop environment episode
T	Maximum inner loop environment episodes
Q	Action-value function
Q^*	Optimal Action-value function
J	Expected total reward function
L	Loss function
f	Objective function
θ	Q-Table parameters
ϵ	The Epsilon in Epsilon-Greedy Strategy
B	Batch size

Table 2.1: Table of notation used

Chapter 3

Background

This chapter introduces three key concepts: Never Give Up (NGU), Prioritized Replay Buffer, and Evolutionary Strategies. These concepts are crucial for understanding our approach and the rationale behind their implementation. While we will partially discuss the relevance of these methods here, a more detailed analysis will be provided in Chapter 4.

3.1 NGU: Never Give Up Algorithm

In 2020, the Google DeepMind Team (Badia et al.) proposed an approach called "Never Give Up" (NGU) [2] that uses curiosity-driven exploration. This method differs from traditional strategies like epsilon-greedy (see Section 2.5), which selects between exploratory and exploitative actions based on a fixed probability. Instead, NGU modifies the reward function to introduce an intrinsic reward, used exclusively during training. This intrinsic reward encourages exploratory behavior by rewarding the agent for engaging with novel and informative aspects of the environment.

Thus, the augmented reward at time step t (r_t) is defined by $r_t = r_t^e + \beta r_t^i$ where r_t^e is the extrinsic (environment) reward, r_t^i is the introduced intrinsic reward, and β is the coefficient that weights the relevance of the intrinsic reward.

In NGU, the intrinsic reward is defined to achieve the following three goals:

1. Rapidly discourage revisiting the same state within a single episode.
2. Slowly discourage revisiting states that have been visited many times across episodes.
3. Ensure that the notion of state excludes aspects of the environment that are unaffected by the agent's actions. (Note: This property is not applicable in this Master's thesis, as the chosen test environments do not contain any aspects that are unaffected by the agent's actions).

To achieve these goals, NGU uses an episodic memory, M that is reset at each episode. This episodic memory stores the list of all the states (or a comparable representation of each state) that has been visited since the beginning of the episode. At each step, the agent computes an episodic intrinsic reward $r_t^{episodic}$ that evaluates how much different the current state is compared to all the states stored in M. The function used in NGU is the following:

$$r_t^{episodic} = \frac{1}{\sqrt{\sum_{y_i \in N_k} K(f(x_t), f(y_i))} + c} \tag{3.1}$$

where $K : R^p \times R^p \to R$ is a kernel function that computes how similar 2 states are, $f(x) : O \to R^p$ is a function that transforms a state to a comparable form, c is a constant that guarantees a min level of similarities (it was fixed to 0.001 by NGU team), and $N_k = \{y_i\}_{i=1,2,...,k}$ are the k-nearest neighbors of x_t in M. The K used by the NGU team was

$$K(x, y) = \frac{\epsilon}{\frac{d^2(x,y)}{d_m^2} + \epsilon} \tag{3.2}$$

where ϵ is a small constant set to 10^{-3}, d is the Euclidean distance, and d_m^2 is a running average of the squared Euclidean distance of the k-th nearest neighbors. This function generalizes very well across different environments. It does not require updates or adjustments to the coefficients for different environments.

Thus, the $r_t^{episodic}$ encourages the agent to visit as much different states as possible within a single episode. It's also important to note that the notion of novelty thus ignores inter-episode interactions.

To achieve the second goal, which is to "slowly discourage revisiting states that have been visited many times across episodes," the NGU method introduces an inter-episodic novelty module. This is accomplished by multiplying the r_t^{episodic} factor by a lifelong curiosity factor, α_t. Various long-term novelty estimators could be used for α_t; however, NGU employs Random Network Distillation (RND) [3]. The RND method uses a random, untrained convolutional network $g : O \to \mathbb{R}^k$, and during training, a predictor network $g' : O \to \mathbb{R}^k$ is trained to predict the outputs of g on all observed inputs. The goal is to minimize the prediction error, defined as $\text{err}(x_t) = \|g'(x_t; \theta) - g(x_t)\|^2$. The modulator α_t is defined by the following expression:

$$\alpha_t = 1 + \frac{\text{err}(x_t) - \mu_e}{\sigma_e} \tag{3.3}$$

where σ_e is the running standard deviation and μ_e is the running mean of $\text{err}(x_t)$. Over time, as the model encounters more observations, it learns to identify these values more accurately, resulting in a gradual decrease in $\text{err}(x_t)$. Consequently, the reward for these familiar interactions decreases.

By combining the inter-episodic novelty module with the episodic novelty module, we obtain the following expression for r_t^i (the intrinsic reward at time step t):

$$r_t^i = r_t^{\text{episodic}} \cdot \min\left(\max\left(\alpha_t, 1\right), L\right) \tag{3.4}$$

where L is the upper bound for the coefficient, which the NGU team set to 5.

Now that we have the reward function, the NGU paper proposes using a set of N values for β: $\{\beta_i\}_{i=0}^{N-1}$. For each training episode, N agents run, each assigned a different value of β. These agents collectively feed a Universal Value Function Approximator (UVFA) [15] $Q(x, a, \beta_i)$ to simultaneously approximate the optimal value function for each of the augmented rewards.

The NGU paper tested and analyzed multiple values for N and β. The best results were achieved with the NGU algorithm when using $N = 32$ agents, demonstrating a more effective balance of exploration and exploitation. The values for β will be discussed later in Section 4.2.5.

The UVFA also facilitates the implementation of a Q-table for $\beta = 0$, which is fully exploitative and can be used for validation and testing purposes.

The NGU approach has demonstrated exceptional results on the Atari-57 benchmark, doubling the performance of the base agent in all hard exploration games. This method ultimately led to the development of Agent57 [13], the first model to outperform human on all Atari-57 games.

3.2 Prioritized Replay Buffer

In the field of reinforcement learning, efficient management of training data is crucial to ensure rapid algorithm convergence and improved agent performance. The replay buffer [10] is a technique for improving this data management, aimed at reducing the temporal correlation between training samples. The replay buffer is a data storage mechanism that enables reinforcement learning agents to retain and reuse past experiences. Past experiences are stored in the form of transitions t_i, where each transition t_i is defined as (s_i, a_i, r_i, s_{i+1}) with s_i the current state, a_i the action chosed by the policy for the current state, r_i the corresponding reward and s_{i+1}.

In the basic Q-learning algorithm, the transition at every time step is used to update the value function immediately (Algorithm 1). With the replay buffer, we initialize a storage system to retain the transitions. Instead of using the current transition to update the Q table, we draw a batch from the replay buffer to update it (Algorithm 2).

Multiple methods exist for sampling transitions from the replay buffer in reinforcement learning. The simplest approach is to randomly select a batch of size B. This method, while straightforward, treats all experiences as equally important,

Algorithm 2 Q-Learning with Replay Buffer

Initialize the value function $Q(s, a, \theta)$ with all Q-values set to 0
Initialize the learning rate α and the discount factor γ
Initialize the replay buffer RB
while not converged **do**
    Get the initial state $s \in S$
    while s **is not** the terminal state **do**
        Select an action $a \in A$ using an ϵ-greedy policy derived from Q
        Execute action a, receive reward r, and observe next state s'
        Store the transition (s, a, r, s') in RB
        $s \leftarrow s'$
    end while
    Sample a batch of transitions B from RB
    Update the value function Q using the batch B
end while

which may not be optimal for learning efficiency.

A more advanced technique is prioritized experience replay, as introduced by Schaul et al. (2015) [14]. In a Prioritized Replay Buffer, each experience is stored with an associated priority. This priority reflects the estimated importance of the experience, typically measured by the magnitude of the temporal-difference (TD) error. The TD error indicates how surprising or informative the experience is, with larger errors suggesting more significant learning opportunities.

During sampling, experiences with higher priorities are more likely to be selected. This ensures that the agent focuses on learning from the most critical experiences, potentially accelerating the learning process. The sampling probability $P(i)$ for an experience i with priority p_i is often given by:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.5}$$

where α determines the level of prioritization. An α of 0 corresponds to uniform random sampling, while higher values of α increase the focus on high-priority experiences.

However, this method introduces bias because the distribution of sampled experiences no longer matches the distribution in the environment. To correct for this, importance sampling weights are used. These weights adjust the updates to ensure that the learning algorithm remains unbiased. The importance sampling weight w_i for an experience i is calculated as:

$$w_i = \left(\frac{1}{N \cdot P(i)} \right)^{\beta} \tag{3.6}$$

where N is the total number of experiences in the replay buffer, and β controls the degree of correction. As β approaches 1, the bias is fully corrected, but at the cost of increased variance in updates.

By leveraging prioritized experience replay, reinforcement learning agents can improve their efficiency by focusing on the most informative experiences, leading to faster and potentially more robust learning.

Algorithm 3 Q-Learning with Prioritized Replay Buffer

Initialize the value function $Q(s, a, \theta)$ with all Q-values set to 0
Initialize the prioritized replay buffer PRB
while not converged **do**
    Get the initial state $s \in S$
    while s **is not** the terminal state **do**
        Select an action $a \in A$ using an ϵ-greedy policy derived from Q
        Execute action a, receive reward r, and observe next state s'
        Compute the priority p_e of the transition (s, a, r, s')
        Store the transition (s, a, r, s') with priority p_e in PRB
        $s \leftarrow s'$
    end while
    Sample a prioritized batch of experiences B from M
    for each experience e in B **do**
        Update the value function Q using experience e and its importance sampling weight w_e
        Recompute and update the priority p_e of experience e in M
    end for
end while

3.3 Evolutionnary algorithms

Evolutionary algorithms [18] are a category of optimization algorithms that take inspiration from the principles of natural selection and genetics. They principally mirror the evolution procedure thereby enhancing the search for complex problem's optimal or close to optimal solutions.

The main advantage of evolutionary algorithms is that they are part of the family of zero-order optimization algorithms, meaning that these algorithms do not require gradient information to perform optimization. In other words, they only need to evaluate the objective function at different points in the search space to guide the search for optimal solutions.

Evolutionary Strategies

Evolutionary Strategies (ES) are a subset of evolutionary algorithms that primarily focus on optimizing continuous parameter spaces. The basic components and operations of ES include :

- **Population:** Every step of the training has to start with a set of candidates.
- **Mutation:** Generating new candidate solutions by perturbing existing ones with a mutation function.
- **Recombination:** Combining information from multiple parents to generate offspring, although some ES variants may rely solely on mutation.
- **Selection:** Choosing the best candidates based on their fitness values to form the next generation.

Specific Evolutionary Strategy Employed

We will draw inspiration from evolutionary strategies for our algorithm, however, we will use a very specific version of these algorithms that does not rely on the mutation system. Instead of having a population of N individuals that evolve over iterations using the operations mentioned above, we will follow the approach

presented in the paper by Tang and Choromanski (2020) [16].

The principle is relatively simple. Throughout the training process, we will have a single individual characterized by a mean μ_t and a variance σ_t. This individual generates a population based on a Gaussian distribution $\mathcal{N}(\mu_t, \sigma^2)$. Based on the results obtained from this population, we update the individual to generate a better population in the next iteration. The ultimate goal is to converge towards an individual capable of generating the best agents.

More details and a clear description of the algorithm are provided in Section 4.3.

Chapter 4

Method

4.1 The general Approach

To minimize the impact of exploration-exploitation trade-off hyperparameters on the performance of Q-learning, we developed a new approach based on two main ideas:

- Using an intrinsic reward to encourage the actors to discover and visit new states, similar to the method used by Badia et al [2] (NGU)
- Implementing a population-based training (PBT) method for the online tuning of the epsilon-greedy search on the actors. [16]

The combination of these methods was chosen because Badia et al.'s intrinsic reward approach has proven effective in hard exploration scenarios, which were previously difficult to navigate. Since our goal is to develop a universal exploration strategy that works across various environments, NGU showed promise as a starting point. However, NGU employs an epsilon-greedy policy for all its agents, and the choice of epsilon value significantly impacts performance. To address this, we implemented PBT (Population-Based Training) for online tuning, which enhances our method by automatically adjusting epsilon values for optimal performance.

4.2 Usage of Intrinsic Reward and adaptation to Q-learning

4.2.1 Reward Function used

Our intrinsic reward function is based on the episodic novelty module proposed by Badia et al. (NGU) and presented in Section 3.1. We use an episodic memory, M, which is unique for each agent and reset at the beginning of each new episode. At each step, the agent adds a comparable representation of the newly visited state to M. This way, M maintains a list of the comparable representations visited during the episode.

We then use this episodic memory to compute an intrinsic reward that encourages the discovery of new states. For each new state visited, we retrieve the k-th nearest neighbors in M and apply the formula from Badia et al. to calculate the episodic intrinsic reward:

$$r_t^{\text{episodic}} = \frac{1}{\sqrt{\sum_{f_i \in N_k} K(f(x_t), f_i)} + c} \quad \text{with} \quad K(x, y) = \frac{\epsilon}{\frac{d^2(x,y)}{d_m^2} + \epsilon} \tag{4.1}$$

where ϵ and c are small constants set to 0.001, d is the Euclidean distance, and d_m^2 is a running average of the squared Euclidean distance of the k-th nearest neighbors. This function normalizes the Euclidean distance across different environments.

We use a function $f(x)$ to transform the state into a comparable representation. In NGU, this representation is computed via deep learning to reduce the impact of environmental objects that do not affect the reward. However, since we address only simple environments in this Master's thesis, we simplify this function to generate a straightforward comparable representation of the state. This function is defined separately for each environment. For instance, in the Wumpus world, it is set to the agent's coordinates on the map.

This episodic reward constitutes our intrinsic reward. We do not use the

inter-episodic novelty module of NGU. Thus, our augmented reward is:

$$r_t = r_t^e + \beta r_t^{\text{episodic}} \tag{4.2}$$

where β quantifies the impact of our intrinsic reward relative to the extrinsic reward.

4.2.2 UVFA & The Need for an alternative

Badia et al. employed a Universal Value Function Approximator (UVFA) to simultaneously learn the Q-values for a set of augmented rewards $r_t^{\beta_i} = r_t^e + \beta_i r_t^i$ during training. However, the complexity of UVFA is greater than that of standard value function approximation [15], and representing a UVFA requires a sophisticated function approximator such as a deep neural network. Our objective is to apply these algorithms to simple Q-learning, and since no existing solution was found in the literature, we had to devise an alternative approach.

4.2.3 Naive Approach

Our first idea was to build a Q-table for each augmented reward (r^{β_i}). Each actor is given a corresponding Q-table and then runs sequentially following an ϵ-greedy policy on its Q-table, and updates the entire Q-table for all augmented rewards ($r_t^{\beta_i}$). Thus, each agent updates not only its own Q-table but also those of the other agents.

Algorithm 4 Naive Approach for NGU implementation to Q-learning

Input: Number of actors: N_{actor}, Number of episodes: T, Learning rate: α, Discount factor γ
$\beta_i \leftarrow InitBetas()$ for $i = 1, 2, ..., N_\beta$ # $N_\beta >= N_{actor}$
$\epsilon^{actor} \leftarrow ChooseEpsilonForEachActor()$ for $actor = 1, 2, ..., N_{actor}$
$Q(s, a, \theta^{(\beta_i)}) \leftarrow InitQTables()$ # One for each value of beta
for $t = 1, 2, \ldots, T$ **do**
    for $actor = 1, 2, \ldots, N_{actor}$ **do**
        $Reset(M)$ # Reset Episodic Memory
        while not Done **do**
            Observe state s.
            $M \leftarrow addComparableRepresentationOf(s)$
            Choose action a using ϵ^{actor}-greedy policy based on $Q(s, a, \theta^{(\beta_{actor})})$.
            Take action a, observe extrinsic reward r^e and next state s'.
            $r^i \leftarrow computeIntrinsicReward(s', M)$
            for each β_i **do**
                $r^{\beta_i} = r^e + \beta_i * r^i$ # augmented reward
                Update $Q(s, a, \theta^{(\beta_i)})$ from transition (s, a, s', r^{β_i})
            end for
            $s \leftarrow s'$
        end while
    end for
end for
return $Q(s, a, \theta^{(\beta=0)})$

It is crucial to maintain and update a Q-table for $\beta = 0$ as it allows for the evaluation phase to disable exploratory behavior entirely. Each actor is assigned a different β_i, ensuring that the number of β values (N_β) equals the number of actors when $N_{\text{actor}} \geq 2$, since at least one actor will run with $\beta_i = 0$. If only one actor is used, N_β equals 2, with one Q-table corresponding to $\beta_i \neq 0$ (the actor's Q-table) and another to $\beta_i = 0$. The specific values chosen for β_i will be discussed in Section 4.2.5.

4.2.4 Approach based on APE-X (PNGU)

A second idea that came to mind was to use a Prioritized Replay Buffer on a distributed learning algorithm inspired from Horgan et al (APE-X) [7]. We refer to this algorithm as PNGU in subsequent sections.

In the method proposed by Horgan et al., multiple actors interact with the environment and add their experiences to a prioritized replay buffer [14] (presented in Section 3.2). A centralized learner then samples from this buffer to update the Q-values. This architecture efficiently leverages distributed learning to enhance performance.

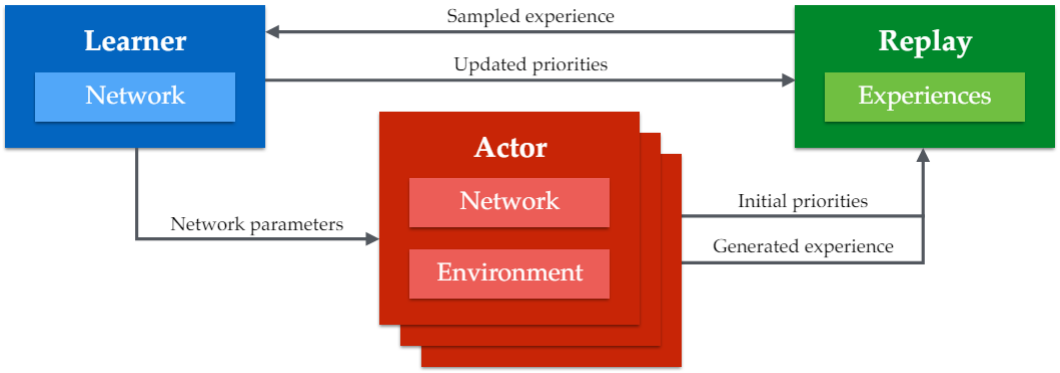


Figure 4.1: APE-X Architecture [7]

To further improve our approach, we incorporated insights from Piot et al. (Agent57) [13], who demonstrated that the state-action value function can be decomposed as follows:

$$Q(x, a, j; \theta) = Q(x, a, j; \theta^e) + \beta_j Q(x, a, j; \theta^i) \tag{4.3}$$

where $Q(x, a, j; \theta)$ is the Q-value function of the augmented reward $r_t^{\beta_j}$, $Q(x, a, j; \theta^e)$ is the extrinsic component, and $Q(x, a, j; \theta^i)$ is the intrinsic component.

To leverage this, we used a General Extrinsic Q-table and a General Intrinsic Q-table. At the start of each episode, we initialize a temporal Q-table for each agent based on the General Extrinsic Q-table and General Intrinsic Q-table using

the equation provided above.

Each agent then runs according to an ϵ-greedy policy on its Q-table. The agent updates its temporal Q-table at each step and adds the experience to the prioritized replay buffer.

At the end of the episode, we sample from the prioritized replay buffer and use this data to update the General Extrinsic Q-table and General Intrinsic Q-table. This approach ensures that the most significant experiences are prioritized during the update process, enhancing the efficiency and effectiveness of learning.

Algorithm 5 PNGU Approach

Input: Number of actors: N_{actor}, Number of episodes: T, Learning rate: α
$\beta_i \leftarrow InitBetas()$ for $i = 1, 2, ..., N_\beta$ # $N_\beta >= N_{actor}$
$Q(s, a, \theta^e) \leftarrow InitGeneralExtrinsicQTable()$
$Q(s, a, \theta^i) \leftarrow InitGeneralIntrinsicQTable()$
$\epsilon^{actor} \leftarrow ChooseEpsilonForEachActor()$ for $actor = 1, 2, ..., N_{actor}$
$PRB \leftarrow initPrioritizedReplayBuffer()$
for $t = 1, 2, \ldots, T$ **do**
    for $actor = 1, 2, \ldots, N_{actor}$ **do**
        $Q(s, a, \theta^{\beta_{actor}}) \leftarrow generateTempQTable(Q(s, a, \theta^e), Q(s, a, \theta^i), \beta_{actor})$
        $Reset(M)$
        while not Done **do**
            Observe state s.
            $M \leftarrow addComparableRepresentationOf(s)$
            Choose action a using ϵ^{actor}-greedy policy based on $Q(s, a, \theta^{(\beta_{actor})})$.
            Take action a on env, observe extrinsic reward r^e and next state s'.
            $r^i \leftarrow computeIntrinsicReward(s', M)$
            $r^{\beta_{actor}} = r^e + \beta_{actor} * r^i$
            Update $Q(s, a, \theta^{(\beta_{actor})})$ from transition $(s, a, s', r^{\beta_{actor}})$
            $p \leftarrow computeExperiencePriority(s, a, s', r^e)$
            Add Experience (s, a, s', r^e) with Priority p to PRB
            $s \leftarrow s'$
        end while
    end for
    Sample a Batch Size of Experiences from the PRB
    for each sampled experience **do**
        Update $Q(s, a, \theta^e)$ and $Q(s, a, \theta^i)$ based on experience
        Recompute & Update priority of used experience in PRB
    end for
end for
return $Q(s, a, \theta^e)$

4.2.5 Values for Betas

Badia et al. trained their Universal Value Function Approximator (UVFA) on a family of augmented rewards characterized by their β_i. This includes the special cases of $\beta_0 = 0$ (full-exploitation) and $\beta_{N-1} = \beta$, where β is the maximum chosen value. They found that using a wider range of β values, beyond just 0 and β, improved performance. To generate the set of β_i, they used the following function:

$$\beta_i = \begin{cases} 0 & \text{if } i = 0 \\ \beta & \text{if } i = N-1 \\ \beta \cdot \sigma\left(10^{\frac{2i-(N-2)}{N-2}}\right) & \text{otherwise} \end{cases}$$

This choice of β_i allows for a focus on the two extreme cases: the fully exploitative policy and the highly exploratory policy.

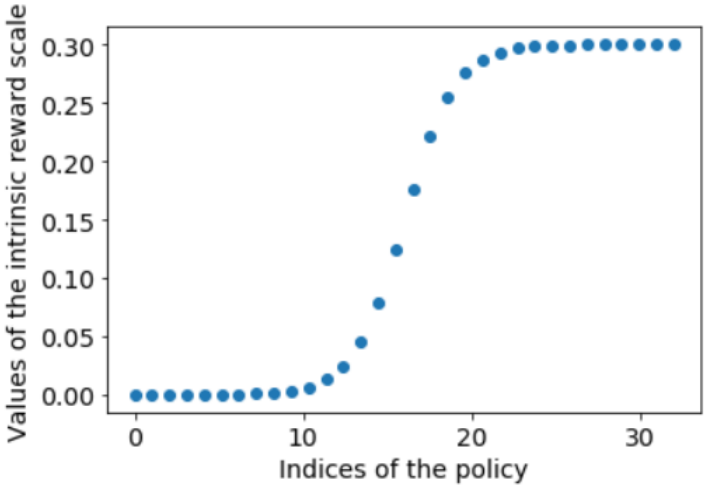


Figure 4.2: Values of the intrinsic reward Scale (β_i) based on Indices of the policy (i), we observe that the function used focus on fully exploitative policy and highly exploratory policy [2]

Badia et al. used a large number of actors, with their best model utilizing $N = 32$ actors. Since we are focusing on Q-learning and not on DQN, and due to the simplifications of UVFA as explained in previous sections, we found that a large number of actors is not necessary. In our experiments, using 1 or 2 actors

yielded better results. (See Appendix B)

4.2.6 Values for Epsilon

As mentioned earlier, each actor operates with its own $\epsilon^{(j)}$-greedy policy. The values for $\epsilon^{(j)}$ must be generated carefully, as they significantly impact performance. Each actor follows a unique β_i policy, so the $\epsilon^{(j)}$ values should be chosen accordingly. Our reasoning is that an actor with highly exploratory behavior (due to a high β_i) should consistently act exploratorily within this policy, which corresponds to a high $\epsilon^{(j)}$. Conversely, an actor with highly exploitative behavior (due to a low β_i) should act more exploitatively within this policy, corresponding to a low $\epsilon^{(j)}$.

To generate the different epsilon values, we employed the generating function established by Horgan et al. [7], but in reverse order according to index j to match the explained reasoning:

$$\epsilon^{(j)} = \epsilon^{1+\alpha\frac{N-1-j}{N-1}} \tag{4.4}$$

In this function, α is a constant set to 8, and ϵ is a parameter representing the maximum epsilon value, typically set to 0.3, though it can vary depending on the environment. The mechanism for tuning this ϵ will be discussed in the subsequent sections.

Additionally, we experimented with using the generating function in reverse order ($\epsilon^{(j)} = \epsilon^{1+\alpha\frac{j}{N-1}}$) and with maintaining a constant epsilon value for all actors ($\epsilon^{(j)} = \epsilon$). Despite these trials, the originally described function consistently yielded the best results.

4.3 Usage of Evolutionary Algorithms on Epsilon

The epsilon parameter in the Q-learning algorithm is typically fixed, presenting a challenge for AutoRL to adapt this parameter dynamically during training. A low epsilon value may be beneficial at the beginning of training but suboptimal towards the end. Our goal is to develop a method capable of dynamically adjusting this

parameter throughout the training process. We propose leveraging evolutionary algorithms and their advantage of being zero-order optimizers.

Evolutionary algorithms are often used to tune a set of parameters by following the procedure described in section 3.3. In our case, our aim was to design an algorithm inspired by evolutionary strategies to tune the epsilon parameter. This type of method has been studied before; Tang and Choromanski [16] propose an algorithm that updates hyper-parameters based on a function f, f representing the agent's performance. Their approach, known as Online Hyper-parameter Tuning via Evolutionary Strategies (OHT-ES), optimizes a smoothed objective function using Gaussian samples, allowing efficient real-time hyper-parameter adjustments. Inspired by their methodology, we applied a similar evolutionary strategy to dynamically adjust the epsilon parameter in epsilon-greedy Q-learning.

4.3.1 Algorithm

Algorithm 6 Online epsilon Tuning for Q-learning via Evolutionary Strategies (EVO-BASIC)

Input: initial mean μ_0 and variance σ^2 for ϵ, number of agents N_{agent}, number of episodes T.
Initialize $Q(s, a; \theta_t^{(n)})$ for $1 \leq n \leq N_{agent}$
Initialize $Q(s, a; \theta)$
for $t = 1, 2, \ldots, T$ **do**
    for $n = 1, 2, \ldots, N_{agent}$ **do**
        while not Done **do**
            Observe state s.
            Choose action a using $\epsilon_k^{(n)}$-greedy policy based on $Q(s, a; \theta_t^{(n)})$.
            Take action a, observe reward r and next state s'.
            Update $Q(s, a; \theta_t^{(n)})$ and $Q(s, a; \theta)$
        end while
        Add estimate performance to $\hat{L}_k^{(n)}$ based on Q_{loss} (Equation 4.5)
    end for
    if $t \mod X = 0$ **then**
        $k = k + 1$
        $\mu_k \leftarrow \mu_k + \alpha^{evo} \frac{1}{\sigma N_{agent}} \sum_{n=1}^{N_{agent}} \hat{L}_k^{(n)} (\epsilon_k^{(n)} - \mu_k)$
        Sample N_{agent} values for $\epsilon : \epsilon_k^{(n)} \sim \mathcal{N}(\mu_k, \sigma^2)$, $1 \leq n \leq N_{agent}$
        $Q(s, a; \theta^{(n)}) \leftarrow Q(s, a; \theta)$, $n = 1, 2, ..., N_{agent}$
    end if
end for
return $Q(s, a; \theta)$

1. **Input:**
    (a) The initial agent parameters θ are the empty Q table for a new training
    (b) μ and variance σ^2 values for the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, σ^2 is fixed and μ is the tuned value.
    (c) Number of agents N_{agent}, number of episodes T.

See Appendix B to see the parameters used in our experiments

2. **Training Loop:**

   At each episode t, we sample N values of ϵ using the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. Then, we initialize a Q-table for each agent with its own parameters, as well as a global Q-table which will be the final Q-table.

   Each agent then runs the episode with its own $\epsilon_t^{(j)}$, updates its Q-table and the global Q-table, and estimates its performance based on the chosen reward function for the meta-objective.

   Every X episodes, we update the mean of the Gaussian distribution, thereby altering the ϵ values sampled at the start of the subsequent episodes. The rationale for updating the mean every X episodes is that a single episode does not provide enough time for an agent to accurately assess its performance. Allowing more episodes helps to limit excessively random behaviors. Further details are provided in the following sections. Additionally, before the next iteration, we synchronize each agent's Q-table with the main Q-table to ensure consistency and merge updates across all agents.

The algorithm leverages the flexibility of evolutionary strategies to adaptively tune the epsilon parameter in the epsilon-greedy policy, potentially improving the performance of the Q-learning algorithm over time by dynamically adjusting to the learning environment's needs.

4.3.2 Meta objective

The meta objective $\hat{L}_t^{(j)}$ represents the agent's performance, forming the basis for the algorithm to adjust the distribution of epsilon. This function significantly influences the results, so defining it accurately is crucial. One of the challenges posed by reinforcement learning algorithms is the indirect visibility of rewards. In environments defined as "hard exploration," the outcome of an action may only become visible after several iterations. This presents a significant challenge because

the reward may become unusable, whereas using the cumulative discounted reward seems intuitive initially.

In our case, the hyperparameter we aim to optimize is epsilon (ϵ), representing the exploration-exploitation trade-off in our Q-learning algorithm. Therefore, instead of using the performance returned by the environment, we opt for the Q-loss:

$$Q_{loss} = abs(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{4.5}$$

This metric represents the impact the agent has had on the Q-table. We have chosen this metric to represent the agents' performance.

4.3.3 Update function

To update the mean of the distribution, we first compute the meta-objective for each agent based on their performance over the last X episodes (with $X = 10$ in our experiments). We then normalize the meta-objective. Once we have the meta-objectives of each agent, we update the average as shown in Algorithm 6:

$$\mu_{k+1} \leftarrow \mu_k + \alpha^{evo} \frac{1}{\sigma N_{agent}} \sum_{n=1}^{N_{agent}} \hat{L}_k^{(n)} (\eta_k^{(n)} - \mu_k) \tag{4.6}$$

This equation is derived from the paper by Tang and Choromanski (2020) [16]. Here, α^{evo} is a pre-defined parameter representing the learning rate, which is set to 0.2 in all our experiments.

4.3.4 Application on NGU and PNGU

Algorithm 7 EVO-NGU

Input: initial mean μ_0 and variance σ^2 for ϵ, number of Agents N_{agent}, number of episodes T, number of actor N_{actor}
$\beta_i \leftarrow InitBetas()$ for $i = 1, 2, ..., N_\beta$ # $N_\beta >= N_{actor}$
$Q^n(s, a, \theta^{(\beta_i)}) \leftarrow InitQTables()$ for each β_i and for $n = 1, 2, ..., N_{agent}$
Initialize $Q(s, a; \theta^{(\beta_i)})$ for each β_i and $k = 0$
Sample N initial values for ϵ : $\epsilon_k^{(n)} \sim \mathcal{N}(\mu_k, \sigma^2)$, $1 \leq n \leq N_{agent}$
for $t = 1, 2, \ldots, T$ **do**
    for $n = 1, 2, \ldots, N_{agent}$ **do**
        $\epsilon^{actor} \leftarrow ChooseEpsilonForEachActor(\epsilon_k^{(n)})$ (See Section 4.2.6)
        for $actor = 1, 2, \ldots, N_{actor}$ **do**
            $Reset(M)$
            while not Done **do**
                Same as NGU using each agent's epsilon ϵ^{actor} to select action
                for each β_i **do**
                    $r^{\beta_i} = r^e + \beta_i * r^i$
                    Update $Q^n(s, a, \theta^{(\beta_i)})$ and $Q(s, a; \theta^{(\beta_i)})$
                end for
            end while
        end for
        Add estimate performance to $\hat{L}_k^{(n)}$ based on Q_{loss} (Equation 4.5)
    end for
    if $t \mod X = 0$ **then**
        $k = k + 1$
        $\mu_k \leftarrow \mu_k + \alpha^{evo} \frac{1}{\sigma N_{agent}} \sum_{n=1}^{N_{agent}} \hat{L}_k^{(n)} (\epsilon_k^{(n)} - \mu_k)$
        Sample N_{agent} values for ϵ : $\epsilon_k^{(n)} \sim \mathcal{N}(\mu_k, \sigma^2)$, $1 \leq n \leq N_{agent}$
        $Q^n(s, a; \theta^{(\beta_i)}) \leftarrow Q(s, a; \theta^{(\beta_i)})$, $n = 1, 2, ..., N_{agent}$
    end if
end for
return $Q(s, a, \theta^{(\beta=0)})$

Algorithm 8 EVO-PNGU

Input: Q-learning function $Q(s, a; \theta)$, initial agent parameters θ, initial mean μ_0 and variance σ^2 for ϵ, number of Agents N_{agent}, number of episodes T, number of actor N_{actor}
$\beta_i \leftarrow InitBetas()$ for $i = 1, 2, ..., N_\beta$ # $N_\beta >= N_{actor}$
$Q(s, a, \theta^e) \leftarrow InitGeneralExtrinsicQTable()$
$Q(s, a, \theta^i) \leftarrow InitGeneralIntrinsicQTable()$
$PrioritizedReplayBuffer \leftarrow initPrioritizedReplayBuffer()$
$k = 0$
Sample N initial values for ϵ : $\epsilon_k^{(n)} \sim \mathcal{N}(\mu_k, \sigma^2)$, $1 \leq n \leq N_{agent}$
for $t = 1, 2, \ldots, T$ **do**
    for $n = 1, 2, \ldots, N_{agent}$ **do**
        $\epsilon^{actor} \leftarrow ChooseEpsilonForEachActor(\epsilon_k^{(n)})$ (See Section 4.2.6)
        for $actor = 1, 2, \ldots, N_{actor}$ **do**
            $Q(s, a, \theta^{\beta_{actor}}) \leftarrow generateTempQTable(Q(s, a, \theta^e), Q(s, a, \theta^i), \beta_{actor})$
            $Reset(M)$
            while not Done **do**
                Same as PNGU
            end while
        end for
        Add estimate performance to $\hat{L}_k^{(n)}$ based on Q_{loss} (Equation 4.5)
    end for
    Sample a Batch Size of Experiences from the Prioritized Replay Buffer
    for each sampled experience **do**
        Update $Q(s, a, \theta^e)$ and $Q(s, a, \theta^i)$ based on experience
        Update priority of used experience in Prioritized Replay Buffer
    end for
    if $t \mod X = 0$ **then**
        $k = k + 1$
        $\mu_k \leftarrow \mu_k + \alpha^{evo} \frac{1}{\sigma N_{agent}} \sum_{n=1}^{N_{agent}} \hat{L}_k^{(n)} (\epsilon_k^{(n)} - \mu_k)$
        Sample N_{agent} values for ϵ : $\epsilon_k^{(n)} \sim \mathcal{N}(\mu_k, \sigma^2)$, $1 \leq n \leq N_{agent}$
    end if
end for
return $Q(s, a, \theta^e)$

As can be seen, the application to Never Give Up (NGU) and PNGU is quite straightforward. The advantage that NGU and PNGU offers is that it inherently manages the exploration/exploitation trade-off by using intrinsic rewards. This can enhance the agents' ability to estimate their performance in hard exploration environments where extrinsic rewards are not readily available.

Chapter 5

Test Environments

In order to test and compare our various algorithms, we need diverse environments to generalize the results. Since Q-learning is a technique that does not work on environments with a continuous observation space, as it requires maintaining a table of (state, action) pairs and thus has a finite number of elements, we had to narrow down our choices. Therefore, we selected three environments with sufficient variety to confront our algorithms with different challenges, we tried to mix hard exploration environments with dense reward environments.

5.1 Wumpus World

The Wumpus environment is a game in which the player finds themselves in a closed square room containing several elements:

- Pits: If the player falls into one, they lose the game.
- The Wumpus: If the player enters the Wumpus' cell, they die.
- Gold: If the player enters this cell and collects the gold, they win the game.
- Empty cells: Nothing happens when the player enters them.

Here is the description of the environment:

Action Space

Discrete(7), shape (1,) in the range $\{0, 6\}$

1. **Turn left (0):** Allows the player to pivot left.
2. **Turn right (1):** Allows the player to pivot right.
3. **Move forward (2):** Allows the player to move within the grid.
4. **Grab (3):** This action only has an effect if used on the gold. It allows the player to collect it and win the game.
5. **Climb (4):** Allows the player to exit the grid. If the player exits without the golds, they lose the game.
6. **Shoot (5):** This action only has an effect if used in front of the Wumpus, allowing the player to kill it.
7. **Wait (6):** Does nothing.

Observation Space

The observation space represents the state available to the algorithm for evaluating the action to take and is defined as an 8-tuple:

(coordinate X, coordinate Y, heading, stench, breeze, glitter, bump, scream) where:

- **Coordinate X, Y:** The coordinates of the agent in the grid.
- **Heading:** The direction the agent is facing.
- **Stench:** Indicates the presence of the Wumpus nearby.
- **Breeze:** Indicates the presence of a pit nearby.
- **Glitter:** Indicates the presence of gold in the current cell.
- **Bump:** Indicates that the agent has hit a wall.
- **Scream:** Indicates whether the Wumpus has been killed (0 for no, 1 for yes).

Reward

The reward is computed as follows:

- +1 for every step taken.
- +1000 when the player dies.
- +500 if the gold is reached.
- +500 if the gold is grabbed.

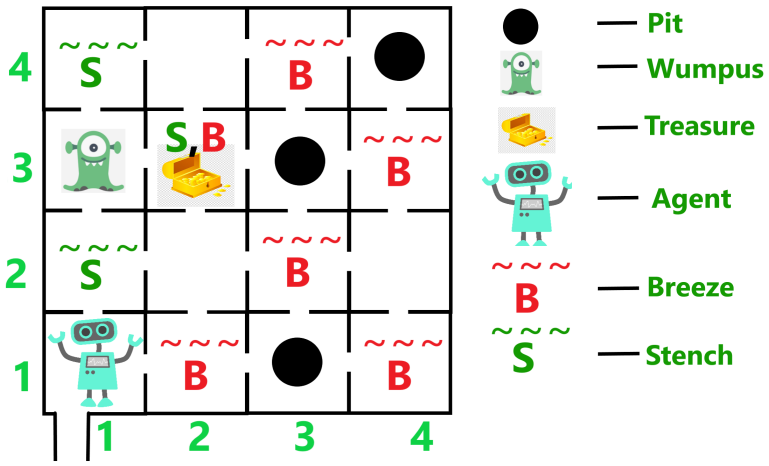


Figure 5.1: Image of the Wumpus game

5.2 Taxi

The Taxi environment in Gymnasium involves navigating to passengers in a grid world, picking them up, and dropping them off at one of four locations.

Action Space

Discrete(6), shape (1,) in the range $\{0, 5\}$

1. **Move south (0)**

2. **Move north (1)**
3. **Move east (2)**
4. **Move west (3)**
5. **Pickup passenger (4)**
6. **Drop off passenger (5)**

Observation Space

The observation space is Discrete(500). An observation is returned as an int() that encodes the corresponding state, calculated by

$$((\text{taxi\_row} \times 5 + \text{taxi\_col}) \times 5 + \text{passenger\_location}) \times 4 + \text{destination}.$$

Reward

The reward is computed as follows:

- +1 for every step taken.
- +20 for delivering the passenger.
- -10 for executing "pickup" and "drop-off" actions illegally.

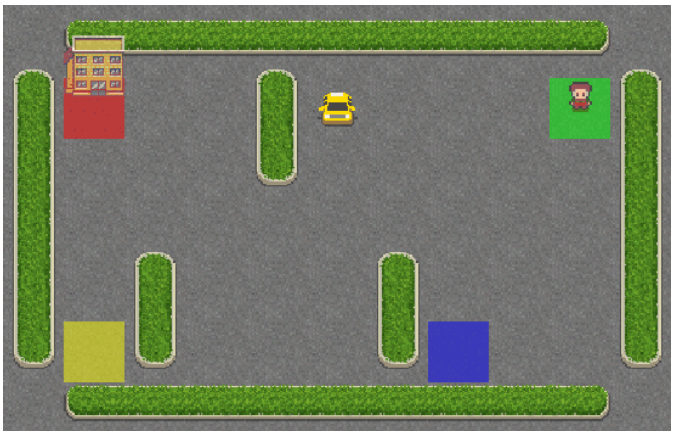


Figure 5.2: Map of the Taxi environment

5.3 Grid World

The Grid World environment involves navigating a grid of size $M \times M$, collecting treasures, avoiding pits, and reaching the exit with the maximum number of treasures.

Action Space

Discrete(4), shape (1,) in the range $\{0, 3\}$

1. **Move up (0)**
2. **Move down (1)**
3. **Move left (2)**
4. **Move right (3)**

Observation Space

The observation space is defined as a tuple $(\text{coorX}, \text{coorY}, \text{treasures\_collected})$, where:

- coorX is the x-coordinate of the player's position.
- coorY is the y-coordinate of the player's position.
- treasures_collected is a list of treasures that have already been collected.

Reward

The reward is computed as follows:

- -1 for every step taken.
- -10 for falling into a pit.
- +10 for collecting a treasure.

- +100 for reaching the exit.

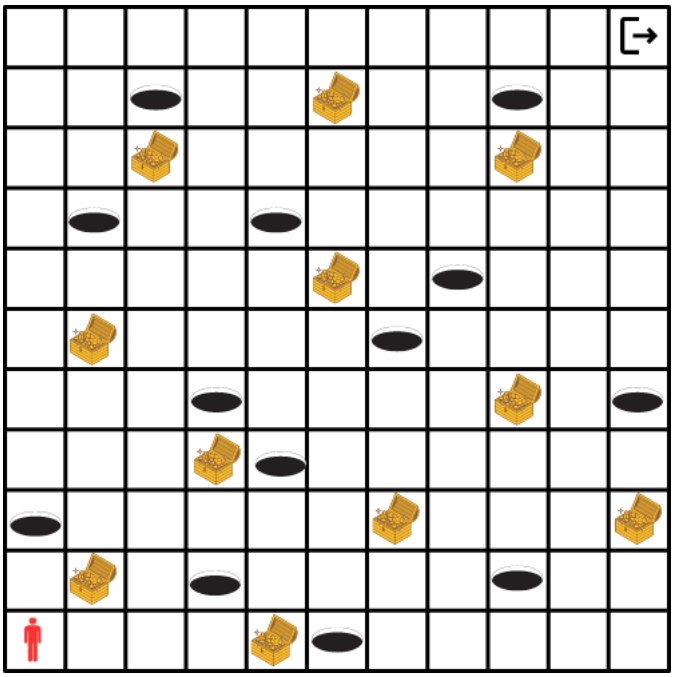


Figure 5.3: Map of the Grid World environment

5.4 FrozenLake

The FrozenLake environment in Gymnasium involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. The player may not always move in the intended direction due to the slippery nature of the frozen lake.

Action Space

Discrete(4), shape (1,) in the range $\{0, 3\}$

1. **Move left (0)**
2. **Move down (1)**

3. **Move right (2)**
4. **Move up (3)**

Observation Space

The observation is a value representing the player's current position as :

$$\text{current\_row} \times \text{nrows} + \text{current\_col}$$

(where both the row and col start at 0). The number of possible observations is dependent on the size of the map.

Reward

The reward schedule:

- Reach goal: +1
- Reach hole: 0
- Reach frozen: 0

Figure 5.4: Map of the FrozenLake environment

Chapter 6

Results & Interpretation

6.1 How do our algorithms perform ?

To evaluate the performance of our various implementations, we compared them to a basic single-actor Q-learning implementation that employs a constant ϵ-greedy strategy. This implementation is referred to as the "Baseline".

It is important to note that the ϵ value for the baseline was determined through hyperparameter tuning, which involved multiple runs of the implementation. In contrast, our evolutionary models (evo-basic, evo-ngu and evo-pngu) dynamically adjusted the ϵ values during runtime, eliminating the need for such tuning.

As we observe on figure 6.1, the implementation of evolutionary algorithms generally diminishes the performance of basic Q-learning (The baseline do better than EVO-BASIC). This is because evolutionary algorithms require multiple agents (here: 3) and it thus implies an increase in the number of environment calls needed to achieve similar results. We also observe that EVO-BASIC is generally less consistent, finding the optimal path after 40,000 environment calls in the best run and 110,000 in the worst run on this environment.

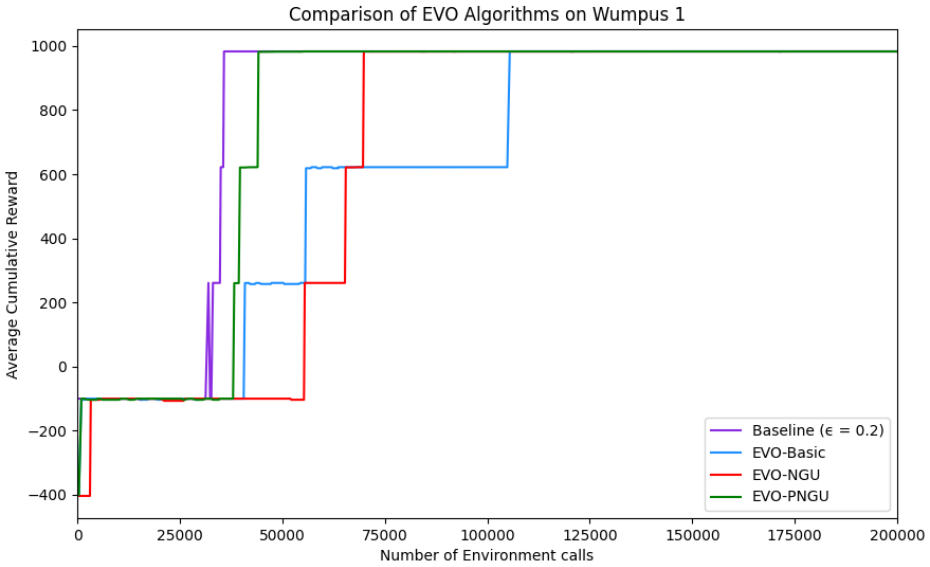


Figure 6.1: Evo Algos on Wumpus 1

The implementation of Naive NGU (EVO-NGU) enhances the performance of EVO-Basic and significantly improves its consistency. This algorithm successfully identified the optimal path in the Wumpus Configuration across all runs after 70,000 environment calls. However, the most impressive results with Evolutionary Algorithms were achieved using our implementation of NGU with Prioritized Replay Buffer and Ape-X (EVO-PNGU). This approach drastically improved performance, nearly matching the baseline and, in some environments, even surpassing it.

When we examine the non-evolutionary algorithms (Figure 6.2), we see that the naive implementation of NGU lowers performance, requiring more environment calls to achieve similar results. Similar to the evolutionary algorithms, this is due to the fact that the naive implementation uses two actors. However, our implementation of NGU based on PRB and Ape-X (PNGU) significantly improves the results, even outperforming the baseline. This behavior is consistent across most environments.

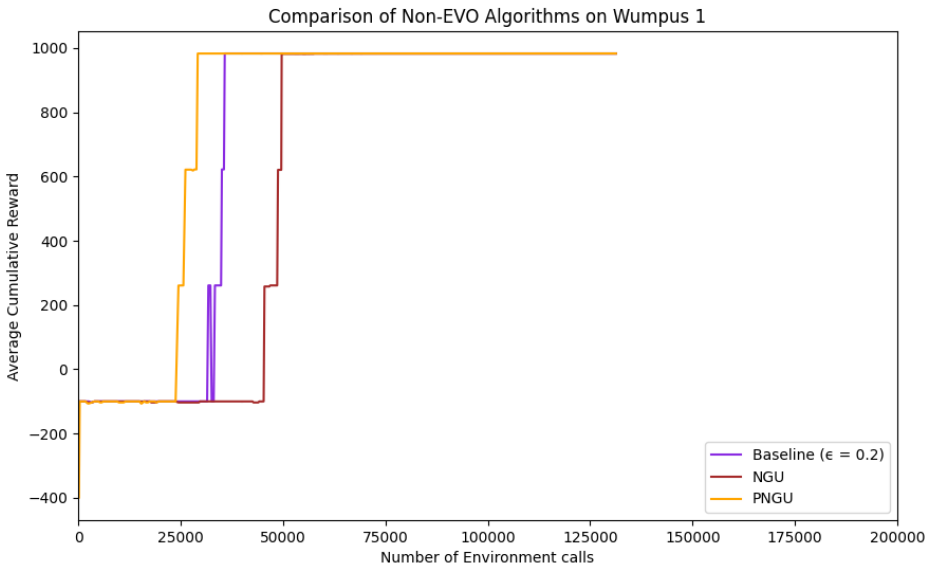


Figure 6.2: Non-Evo Algos on Wumpus 1

Detailed results across all environments are provided in Appendix A, and the selected hyperparameters are listed in Appendix B.

6.2 How well do they adapt to new environments ?

To evaluate how well our algorithms adapt to new environments, we tested their performance on the various environments described in Section 5. Hyper-parameter engineering was applied only to the baseline to identify an optimal learning rate and epsilon-greedy value for each environment. This epsilon-greedy value was then used for all non-evolutionary algorithms. To avoid bias, the initial epsilon value for evolutionary algorithms was set to 0.5, allowing the algorithms to autonomously adjust it during training.

As we will demonstrate in the following sections, our algorithms performed well across all environments. Although some inconsistencies were observed in certain runs, the overall trend was positive.

6.2.1 Frozen Lake

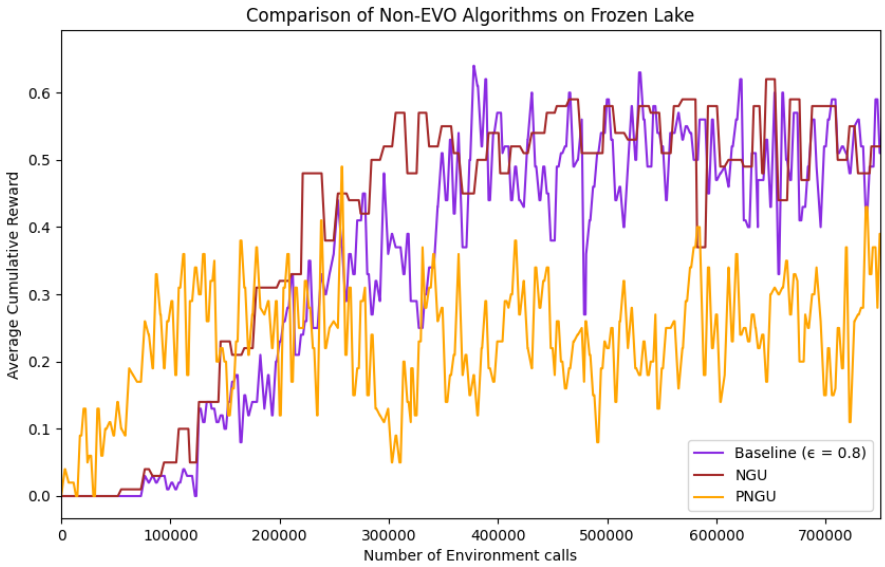


Figure 6.3: Comparison of Non-EVO Algorithms on Frozen Lake

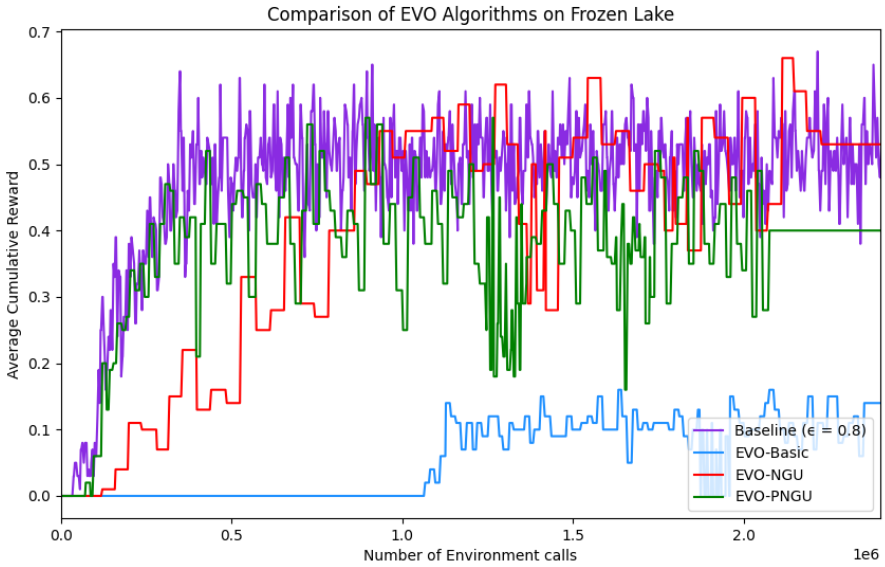


Figure 6.4: Comparison of EVO Algorithms on Frozen Lake

As we can see from the results of the non-evolutionary algorithms, NGU positively impacts the Frozen Lake environment by quickly identifying good paths. Both the Naive NGU and PNGU implementations outperformed the baseline in terms of the number of environment calls required to achieve an acceptable reward score (0.3). Notably, the PNGU implementation demonstrated superior performance compared to the Naive NGU implementation. However, it is noteworthy that PNGU converged to a suboptimal path and failed to achieve the scores of the Baseline and Naive NGU. This suggests that PNGU might sometimes have a tendency to converge prematurely.

For the evolutionary algorithms, we observed the expected behavior as previously described. The EVO-BASIC algorithm performed poorly, while EVO-NGU showed a significant improvement over EVO-BASIC. EVO-PNGU outperformed all other evolutionary models and nearly matched the baseline performance.

6.2.2 Taxi

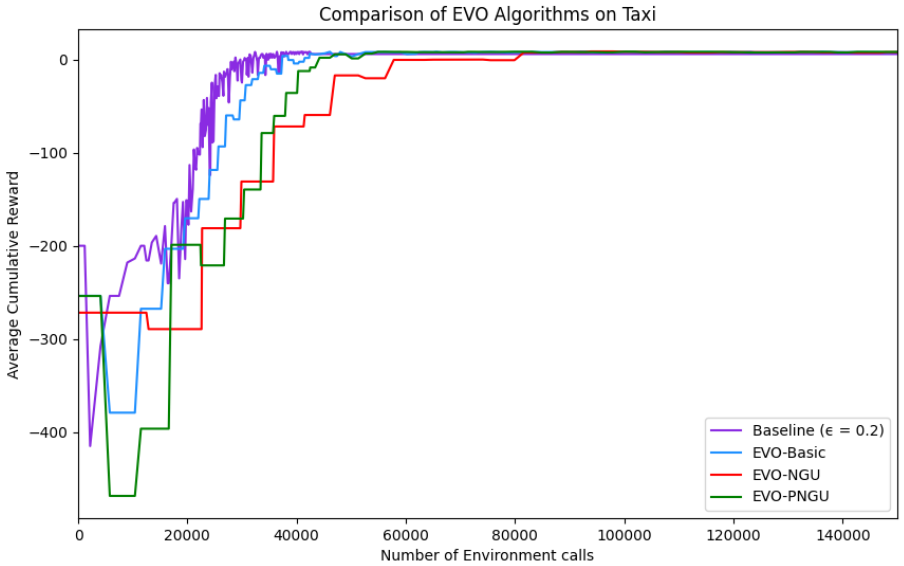


Figure 6.5: Comparison of EVO Algorithms on Taxi

Taxi is the only environment where our implementation showed negative performance. Upon analyzing the reasons, we found that it is due to the environment being very small and easy, where the optimal path is found in fewer than 100 episodes. Since our evolutionary algorithms adapt the epsilon value every 10 episodes, there were only 10 opportunities for epsilon adaptation before convergence, which is too few. Consequently, the epsilon did not have enough time to find the optimal value, leading to most episodes occurring with suboptimal epsilon values, resulting in poor performance.

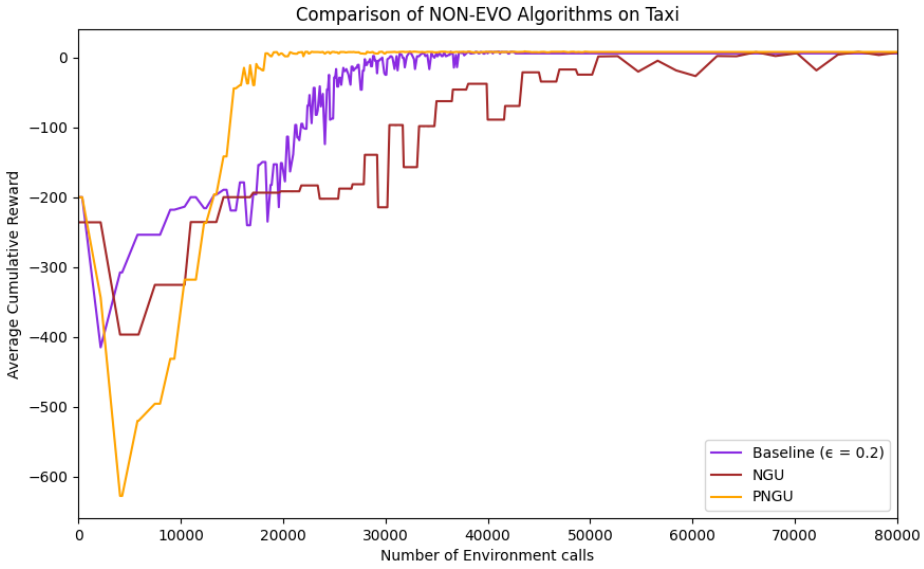


Figure 6.6: Comparison of Non-EVO Algorithms on Taxi

However, we observe excellent results with the non-evolutionary algorithms. They exhibit the expected behavior previously described. The Naive NGU performs worse compared to the baseline, but the PNGU significantly outperforms all other algorithms.

6.2.3 Wumpus

On the Wumpus environment, we tested six different configurations with varying levels of complexity and observed consistent behavior across all tests. Since the

results on tis environment has already been discussed in Section 6.1, we will not delve further into it here. Detailed results for all six configurations can be found in Appendix A.

6.2.4 Grid World

In the Grid World, a dense-reward scenario, our algorithm did not perform as well. This indicates that while our approach works particularly well in sparse-reward scenarios, it struggles to achieve similar success in dense-reward environments. Although both our Naive NGU and EVO-NGU algorithms performed better than the baseline, our PNGU and EVO-PNGU implementations were suboptimal. The reasons for this performance gap and potential areas for improvement are discussed in Chapter 7.3.

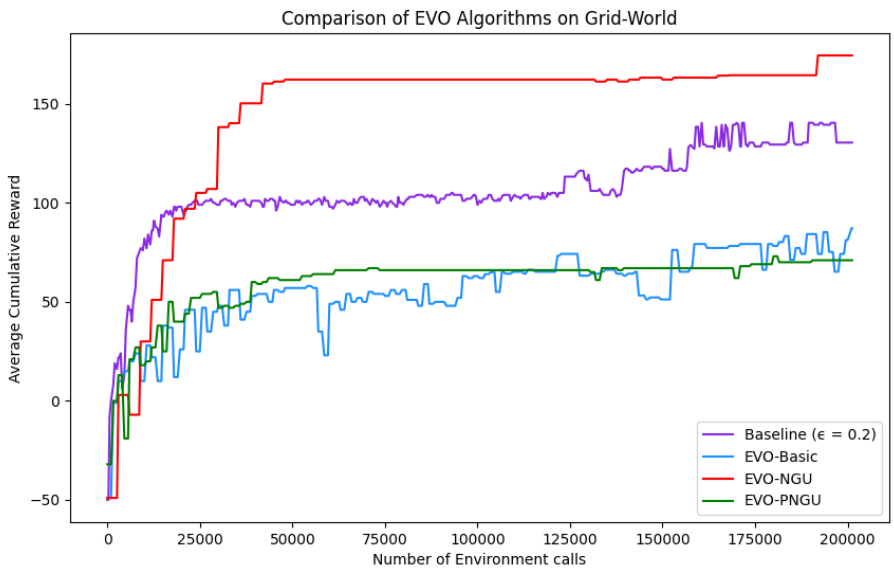


Figure 6.7: Comparison of EVO Algorithms on Grid World

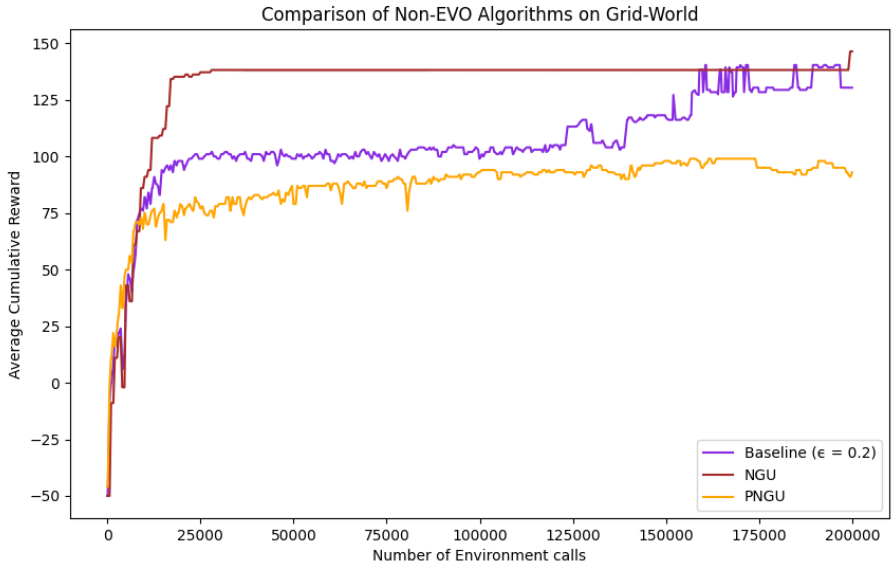


Figure 6.8: Comparison of Non-EVO Algorithms on Grid World

6.3 How does the Epsilon Evolves during training ?

We will look at the evolution of the epsilon parameter across various environments to observe and analyze the behavior of our algorithms. This analysis will help us understand how changes in epsilon influence performance and facilitate explanations of the observed outcomes.

It is important to note that we started each experiment with an epsilon value of 0.5 to maintain neutrality. Our goal was to allow the algorithm to adapt on its own without any external intervention. This approach means that the algorithm takes longer to find the optimal solution compared to starting with a predefined optimal epsilon. However, we believe it is crucial not to influence the results with initial settings that may vary across different environments. This choice aligns with our objective of achieving generalization.

6.3.1 Wumpus world

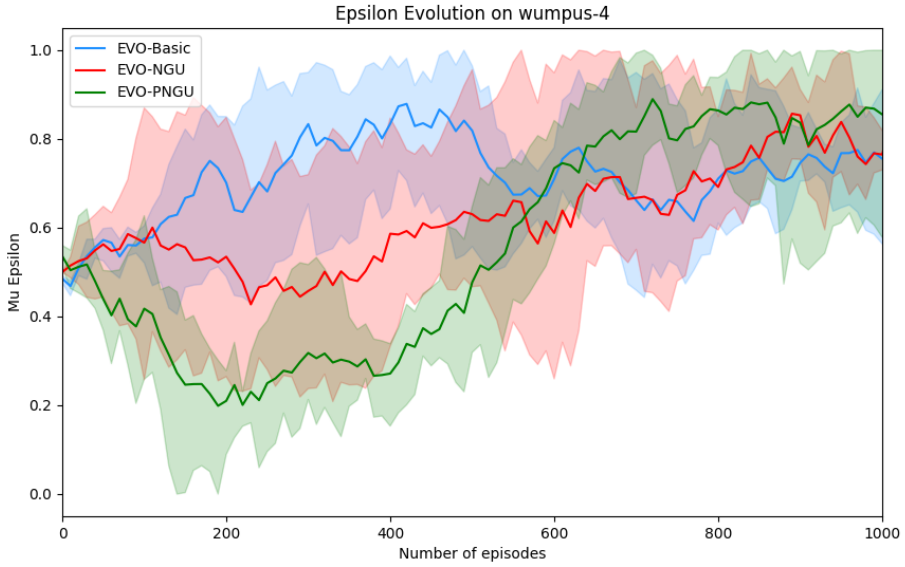


Figure 6.9: Epsilon value during training

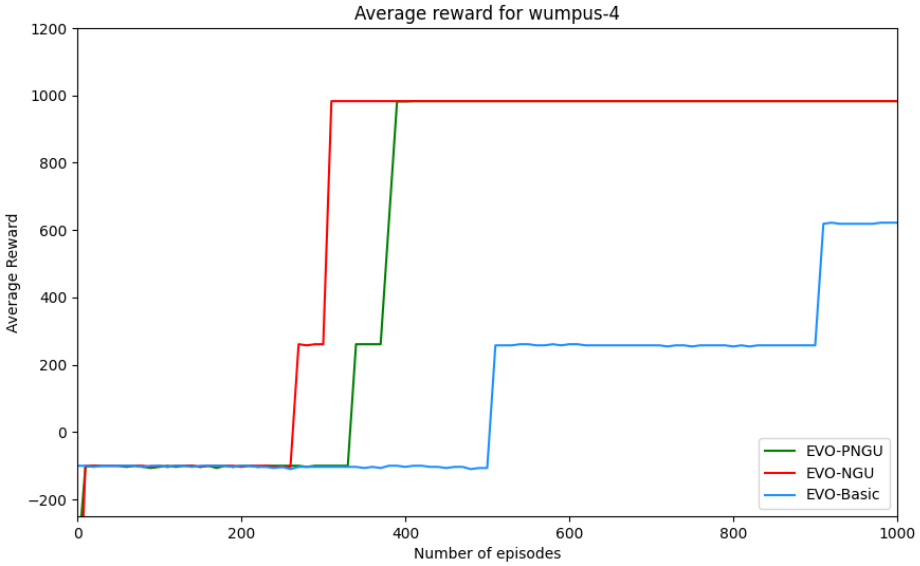


Figure 6.10: Average reward over episodes

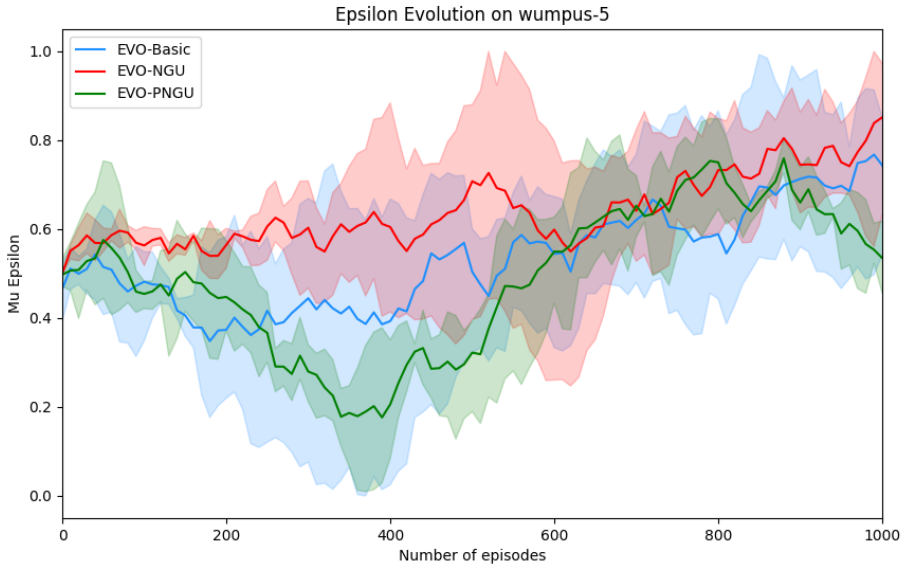


Figure 6.11: Epsilon value during training

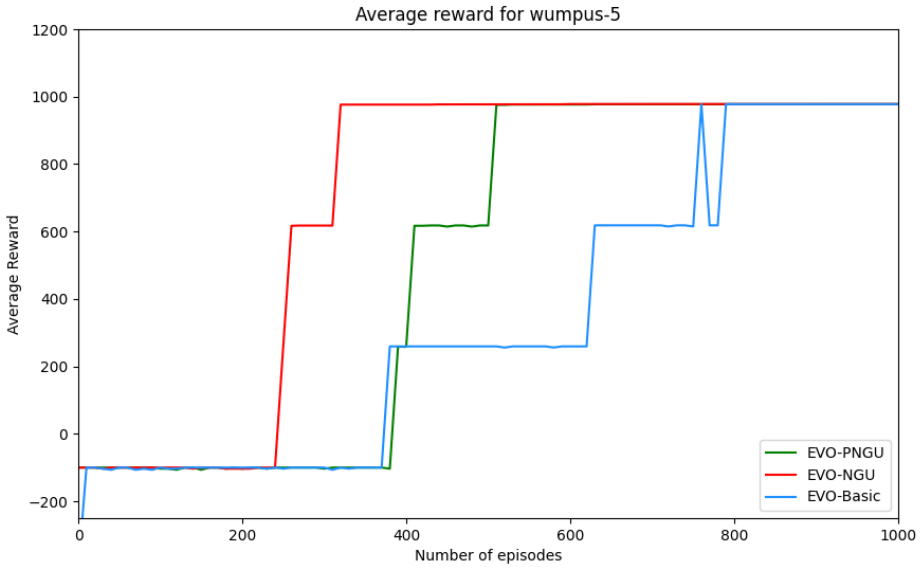


Figure 6.12: Average reward over episodes

It should be noted that the optimal constant epsilon for the Wumpus game is close to 0, determined through a standard grid search.

As seen in the graphs above (Figures 6.9, 6.10, 6.11, 6.12), EVO-PNGU tends to reduce epsilon significantly at the beginning of training, which aligns with the optimal epsilon strategy. However, once it finds the treasure and receives the reward, it tends to increase epsilon again. This behavior was consistently observed across all Wumpus configurations. Initially, by finding a path to the treasure, even if sub-optimal, it then explores other possibilities extensively, increasing the probability of discovering a better path. This results in quickly finding a path to the treasure and subsequently searching for better paths.

This behavior is slightly visible in NGU but much less pronounced. As shown in Figure 6.12, EVO-Basic does not follow this tendency in several configurations and exhibits a much more random behavior.

This demonstrates the contribution of NGU and PNGU in enhancing the evolutionary algorithm's ability to find the best epsilon values during training.

6.3.2 Taxi

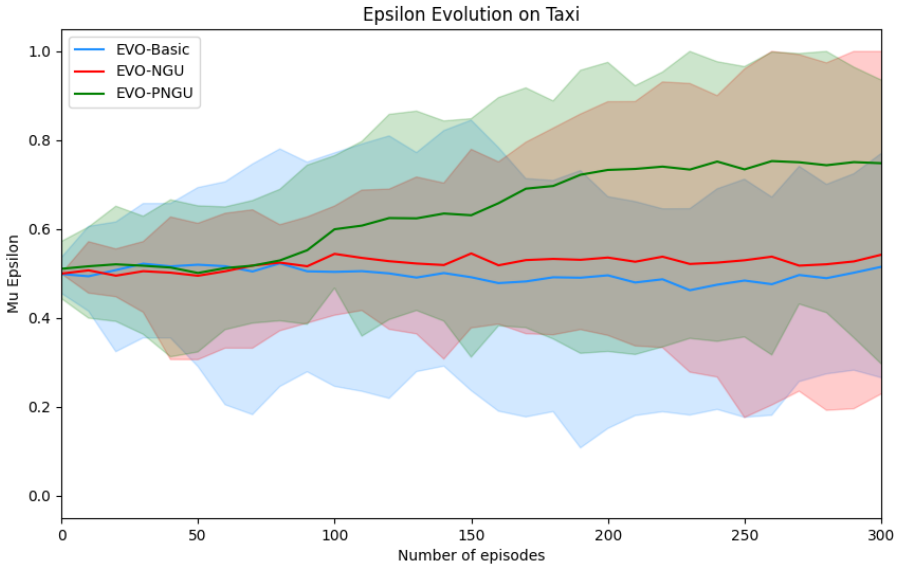


Figure 6.13: Epsilon value during training

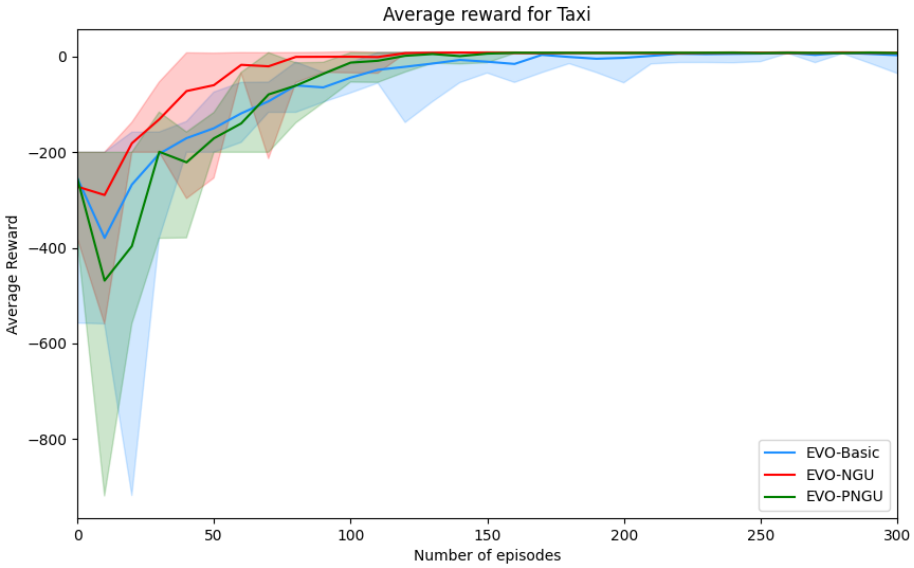


Figure 6.14: Average reward over episodes

In this small testing environment, even basic Q-learning can easily succeed, which may limit the scope of our analysis. However, this scenario also highlights the strengths of EVO-PNGU. As seen in the graph 6.13, EVO-PNGU, after finding the optimal value around 100 episodes, immediately starts increasing its epsilon to explore further, aiming to discover new states that yield better results. Conversely, EVO-NGU and EVO-Basic appear unable to converge and remain at their initial values.

6.3.3 FrozenLake

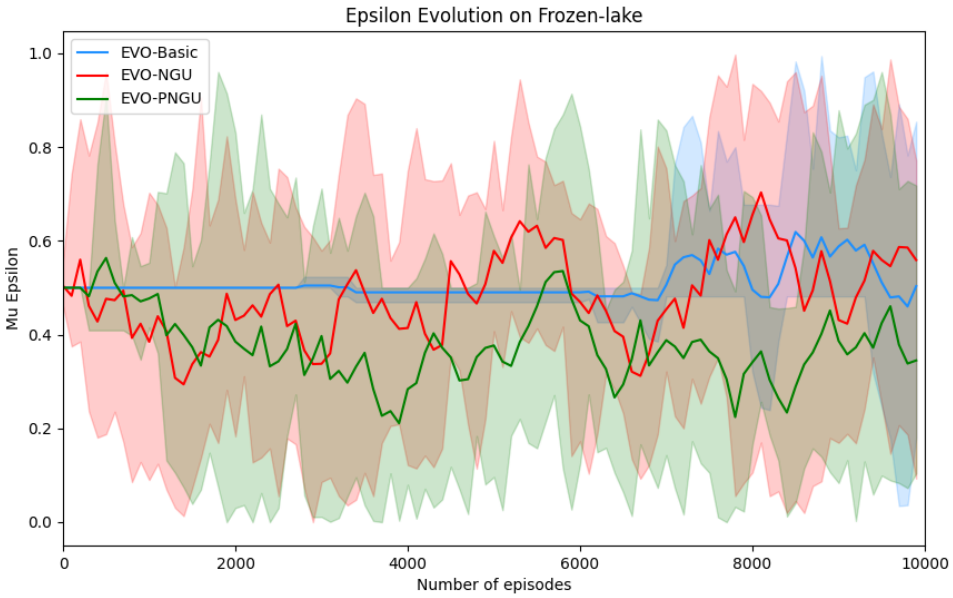


Figure 6.15: Epsilon value during training

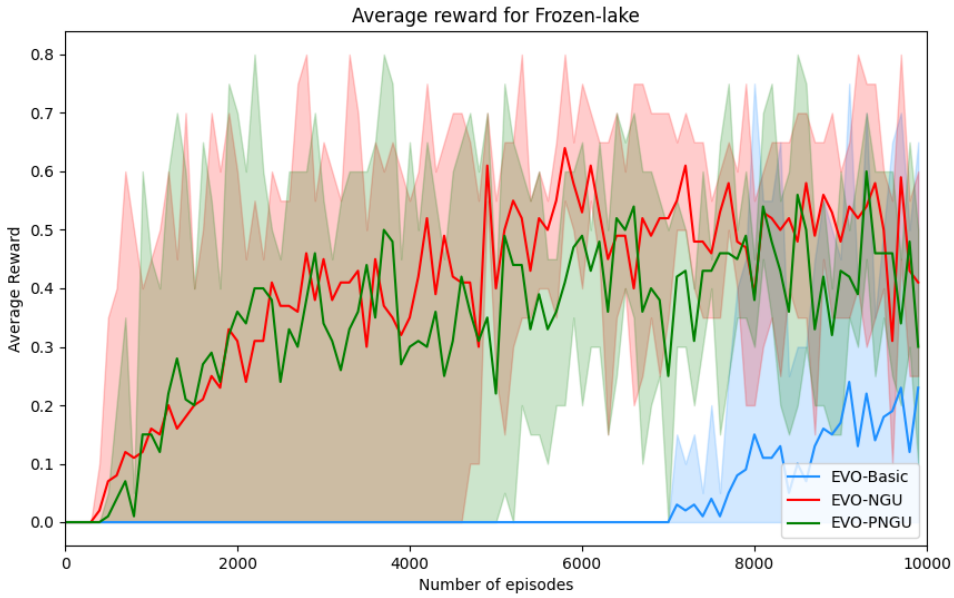


Figure 6.16: Average reward over episodes

FrozenLake presents a unique challenge as each action has a probability of resulting in a random action (as explained in Chapter 5). This adds difficulty for our algorithm in evaluating its performance and converging accordingly. This is evident in the graphs, where the evolution of epsilon appears to stagnate at its initial value. However, it is notable that EVO-PNGU seems to be converging towards lower values, favoring exploitation. This can be explained by the fact that the agent's evaluation is based on loss. Since the actions can result in different outcomes, using the values already contained in its table, it is logical that the loss is more impacted when exploiting rather than exploring. In other words, repeating the same actions has a greater impact due to the variability in outcomes, pushing the agent towards this strategy.

6.3.4 Grid-world

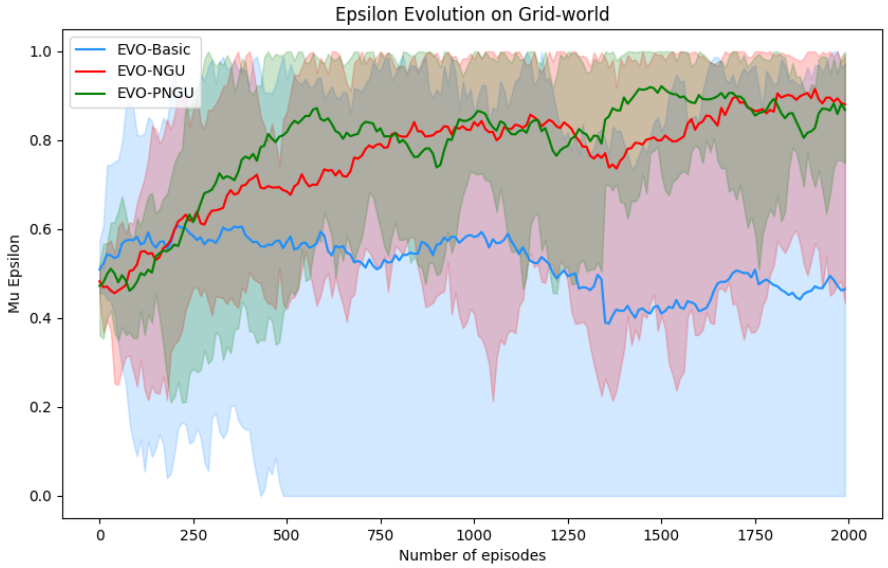


Figure 6.17: Epsilon value during training

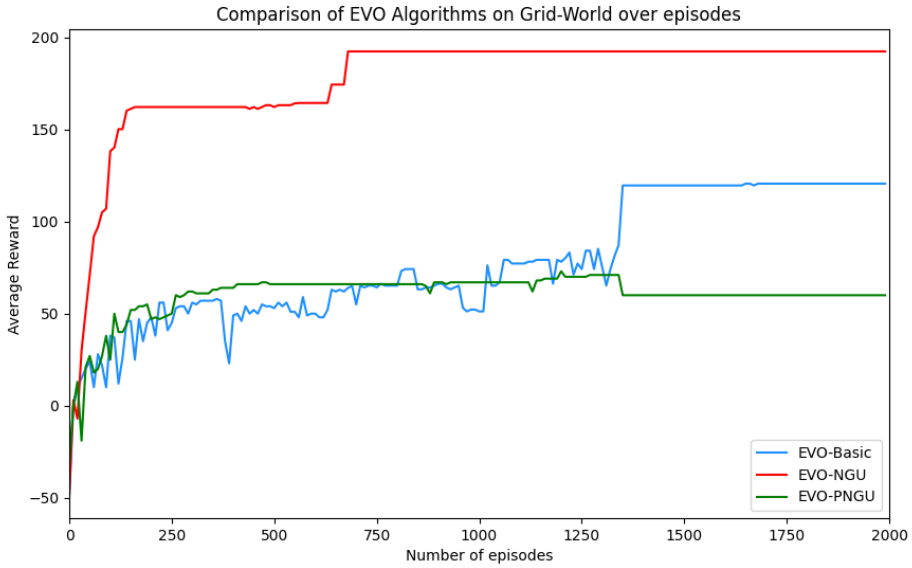


Figure 6.18: Average reward over episodes

As explained, the grid-world environment is characterized by dense rewards. Here, we observe that for NGU and PNGU, the epsilon tends to increase, whereas EVO-Basic stagnates at the initial value. This indicates that NGU and PNGU, even in such environments, can influence the evolutionary algorithm to converge towards a specific epsilon. For NGU, this approach seems to work perfectly, while PNGU faces more challenges. In the following section, we will discuss an improvement that could address this issue.

Chapter 7

Limitations & further research

In this section, we will discuss some limitations we encountered in our algorithms and identify areas for further research and improvement of our model.

7.1 From Basic Q-learning to Deep Q Learning

One of the major challenges in this master's thesis was adapting algorithms developed for Deep Reinforcement Learning to simple Q-learning. This required the development of new techniques, such as the Ape-X implementation of the NGU model. However, these techniques are imperfect, often substituting for the Universal Value-Function Approximator (UVFA), which is complex and requires an advanced function approximator like DQN. This limitation also confined our experiments to simple test environments, even though our models had the potential to perform well in more complex settings.

7.2 Usage of long-term inter-episodic novelty module

As discussed in the presentation of NGU, Badia et al. introduced the episodic novelty module, which we incorporated into our implementation. They also presented a long-term inter-episodic novelty module, which they deemed more optional. Despite its optional status, implementing this module could significantly enhance

our model. However, this implementation requires Random Network Distillation (RND), which we found to be impractical for our goal of developing a simple exploration-exploitation strategy using Q-learning. Including this module would have made the training process more complex and computationally expensive. Nonetheless, if we were to implement Deep Q-learning, incorporating this module would be a logical step.

7.3 Meta-Objective of the evolutionary algorithms

In our method, we employed the sum of the loss on the Q-learning table as the meta-objective for all our algorithms. This approach proved highly effective in sparse-reward scenarios. However, as demonstrated in the Grid World scenario, it was suboptimal in dense-reward scenarios. Consequently, we experimented with other meta-objectives, finding that the cumulative reward was particularly effective for dense-reward scenarios.

A promising area for future research would be to develop a general weighted combination of the loss and cumulative reward. A naive approach, however, would likely fail to generalize across all types of scenarios as Sparse-reward scenarios should prioritize the loss, while dense-reward scenarios should emphasize the cumulative reward. Therefore, an interesting research direction would be to identify a meta-learning technique capable of dynamically learning and adjusting these weights.

Chapter 8

Conclusion

In this master's thesis, we developed and evaluated a new exploration strategy for Q-learning by combining evolutionary algorithms with curiosity-driven approaches. Our primary goal was to create a robust and adaptive exploration-exploitation strategy that performs well across diverse environments without requiring extensive hyper-parameter tuning. We achieved this by integrating the Never Give Up (NGU) method with online tuning mechanisms inspired by Automated Reinforcement Learning (AutoRL).

Our findings indicate that incorporating curiosity-driven intrinsic rewards and evolutionary strategies significantly enhances the performance of standard Q-learning algorithms, particularly by eliminating the need for grid search or other hyper-parameter fine-tuning techniques. Furthermore, we demonstrated that this combination outperforms the simple implementation of evolutionary algorithms on Q-learning.

However, we identified several limitations in our approach that highlight areas for further improvement. These include integrating Deep Reinforcement Learning, implementing a long-term inter-episodic novelty module, and developing a more sophisticated meta-objective that effectively balances performance in both sparse-reward and dense-reward scenarios.

Practically, this work contributes to the field of Automated Reinforcement Learning (AutoRL) by providing a general exploration strategy that does not require hyper-parameter tuning. This advancement has the potential to improve the efficiency and applicability of reinforcement learning algorithms in various real-world applications, making reinforcement learning more accessible to practitioners who are not experts in the field.

Bibliography

[1] Marcin Andrychowicz et al. "What Matters for On-Policy Deep Actor-Critic Methods? A Large-Scale Study". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=nIAxjsniDzg.

[2] Adrià Puigdomènech Badia et al. *Never Give Up: Learning Directed Exploration Strategies*. 2020. arXiv: 2002.06038 [cs.LG].

[3] Yuri Burda et al. *Exploration by Random Network Distillation*. 2018. arXiv: 1810.12894 [cs.LG].

[4] Logan Engstrom et al. "Implementation Matters in Deep RL: A Case Study on PPO and TRPO". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=r1etN1rtPB.

[5] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021), p. 106622. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2020.106622. URL: https://www.sciencedirect.com/science/article/pii/S0950705120307516.

[6] Peter Henderson et al. "Deep Reinforcement Learning That Matters". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 2018). DOI: 10.1609/aaai.v32i1.11694. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11694.

[7] Dan Horgan et al. *Distributed Prioritized Experience Replay*. 2018. arXiv: 1803.00933 [cs.LG].

[8] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[10] Long-Ji Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8 (1992), pp. 293–321.

[11] Charles Packer et al. *Assessing Generalization in Deep Reinforcement Learning*. 2019. arXiv: 1810.12282 [cs.LG].

[12] Jack Parker-Holder et al. "Automated Reinforcement Learning (AutoRL): A Survey and Open Problems". In: *Journal of Artificial Intelligence Research* 74 (June 2022), pp. 517–568. ISSN: 1076-9757. DOI: 10.1613/jair.1.13596. URL: http://dx.doi.org/10.1613/jair.1.13596.

[13] Piot et al. "Agent57: Outperforming the Atari Human Benchmark". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 507–517. URL: https://proceedings.mlr.press/v119/badia20a.html.

[14] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

[15] Tom Schaul et al. "Universal Value Function Approximators". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1312–1320. URL: https://proceedings.mlr.press/v37/schaul15.html.

[16] Yunhao Tang and Krzysztof Choromanski. *Online Hyper-parameter Tuning in Off-policy Learning via Evolutionary Strategies*. 2020. arXiv: 2006.07554 [cs.LG].

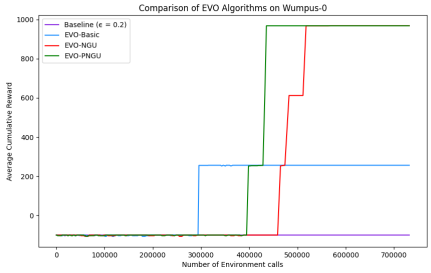
[17] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

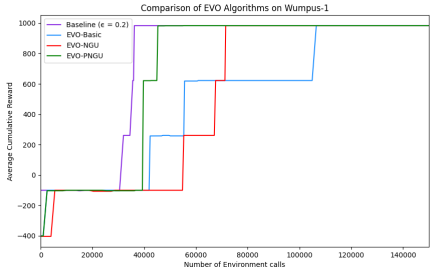
[18] Xinjie Yu and Mitsuo Gen. *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010.

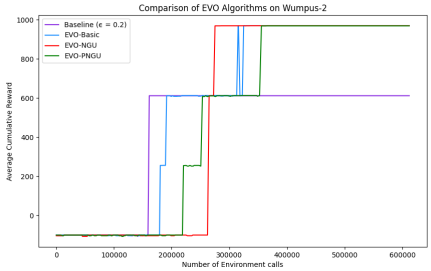
We also utilized Generative AI (ChatGPT) to assist in the writing and refinement of this Master's Thesis. However, it was not used for conducting the research itself.

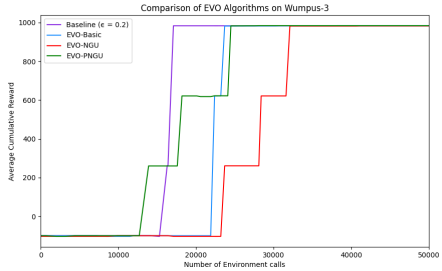
Appendix A

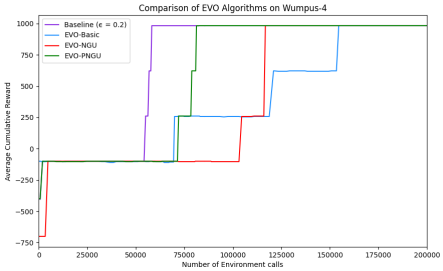
Results Across All Environments

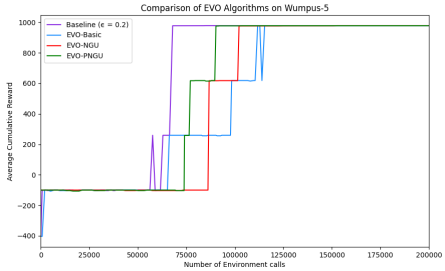


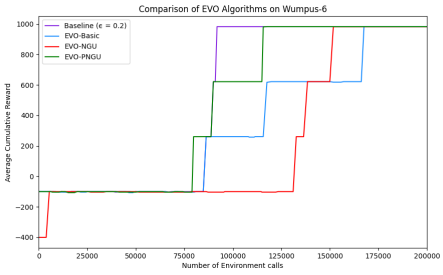


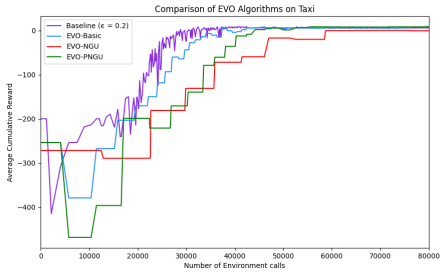


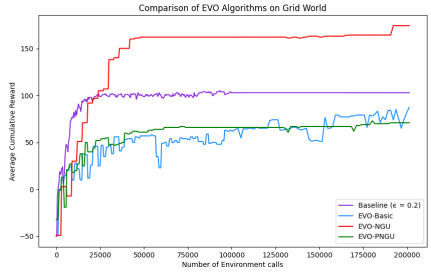


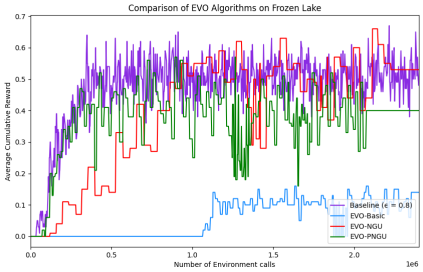


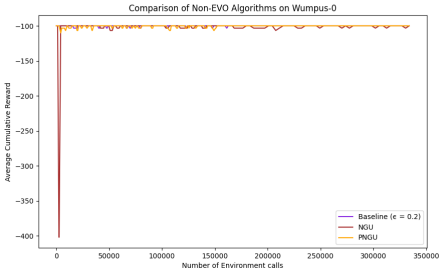


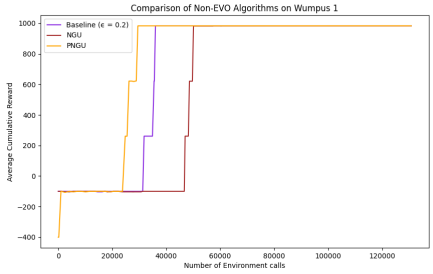

Comparison of EVO Algorithms on Wumpus-6
Baseline (ε = 0.2)
EVO-Basic
EVO-NGU
EVO-PNGU
Average Cumulative Reward
Number of Environment calls

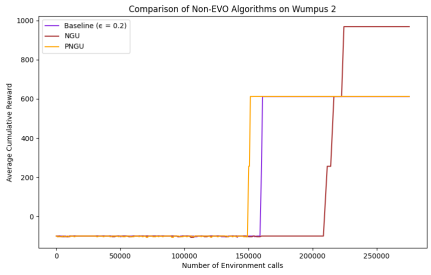

Comparison of EVO Algorithms on Taxi
Baseline (ε = 0.2)
EVO-Basic
EVO-NGU
EVO-PNGU
Average Cumulative Reward
Number of Environment calls

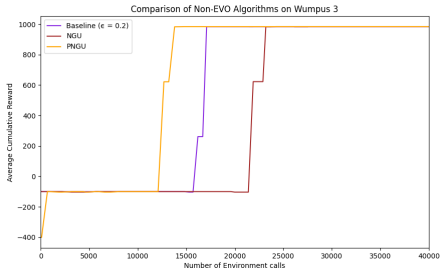

Comparison of EVO Algorithms on Grid World
Baseline (ε = 0.2)
EVO-Basic
EVO-NGU
EVO-PNGU
Average Cumulative Reward
Number of Environment calls

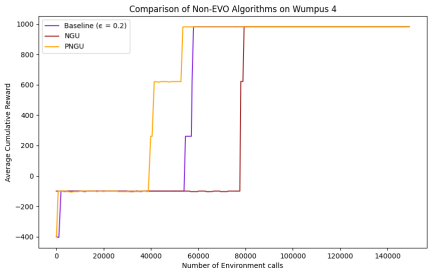

Comparison of EVO Algorithms on Frozen Lake
Baseline (ε = 0.8)
EVO-Basic
EVO-NGU
EVO-PNGU
Average Cumulative Reward
Number of Environment calls
1e6

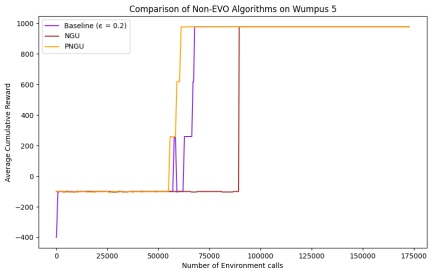

Comparison of Non-EVO Algorithms on Wumpus-0
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

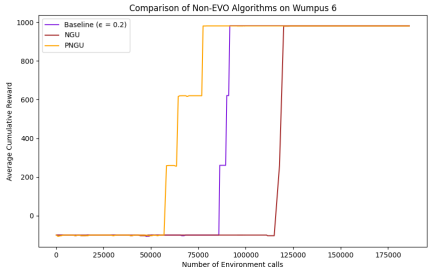

Comparison of Non-EVO Algorithms on Wumpus 1
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

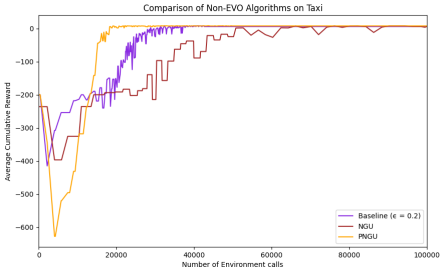

Comparison of Non-EVO Algorithms on Wumpus 2
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

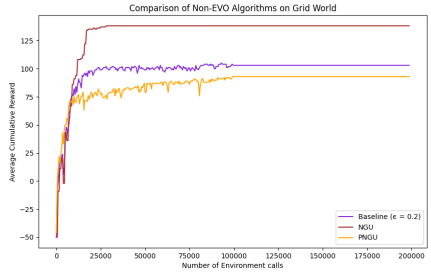

Comparison of Non-EVO Algorithms on Wumpus 3
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

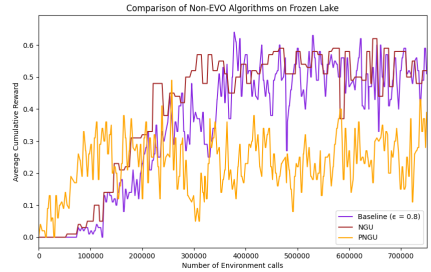

Comparison of Non-EVO Algorithms on Wumpus 4
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

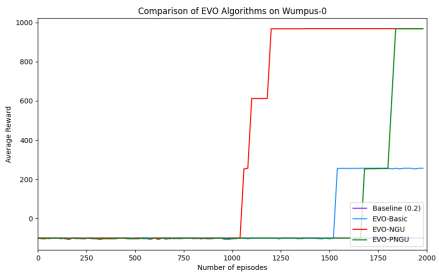

Comparison of Non-EVO Algorithms on Wumpus 5
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

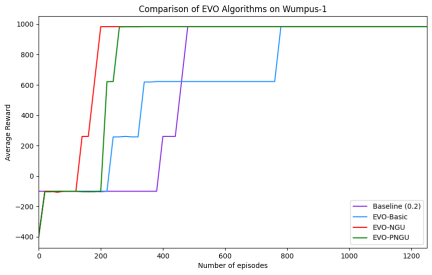

Comparison of Non-EVO Algorithms on Wumpus 6
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

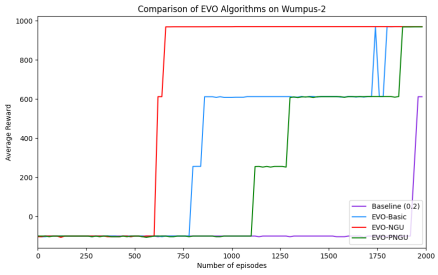

Comparison of Non-EVO Algorithms on Taxi
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

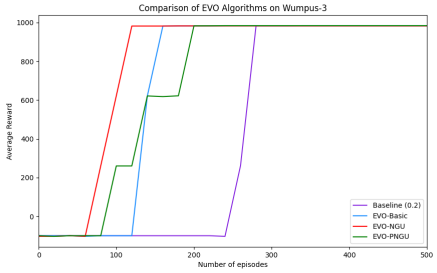

Comparison of Non-EVO Algorithms on Grid World
Baseline (ε = 0.2)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

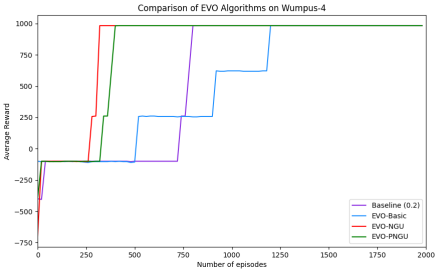

Comparison of Non-EVO Algorithms on Frozen Lake
Baseline (ε = 0.8)
NGU
PNGU
Average Cumulative Reward
Number of Environment calls

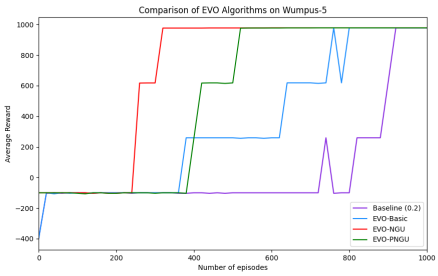

Comparison of EVO Algorithms on Wumpus-0
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU
Average Reward
Number of episodes

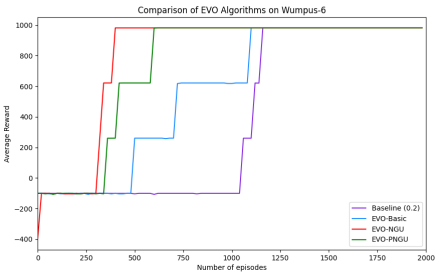

Comparison of EVO Algorithms on Wumpus-1
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

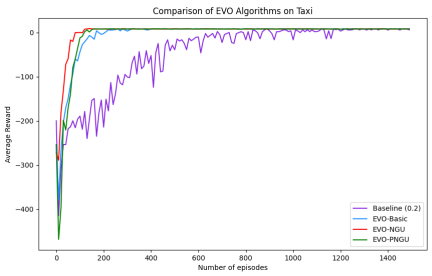

Comparison of EVO Algorithms on Wumpus-2
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

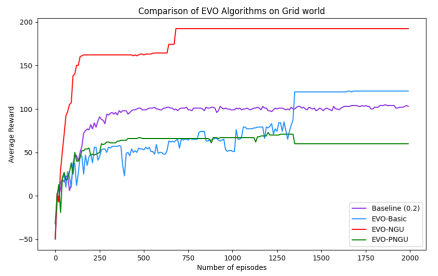

Comparison of EVO Algorithms on Wumpus-3
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

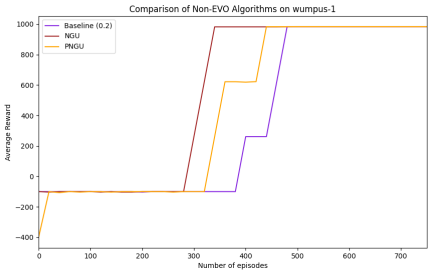

Comparison of EVO Algorithms on Wumpus-4
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

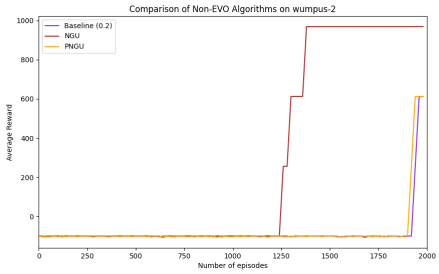

Comparison of EVO Algorithms on Wumpus-5
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

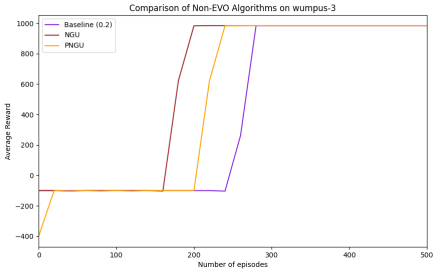

Comparison of EVO Algorithms on Wumpus-6
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

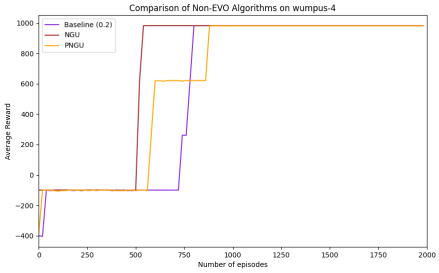

Comparison of EVO Algorithms on Taxi
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

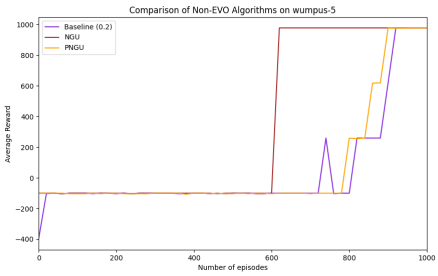

Comparison of EVO Algorithms on Grid world
Average Reward
Number of episodes
Baseline (0.2)
EVO-Basic
EVO-NGU
EVO-PNGU

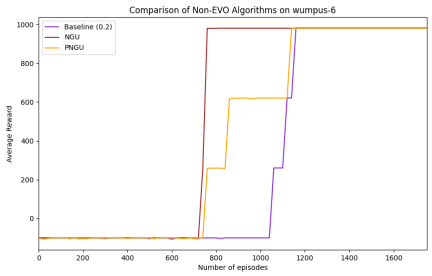

Comparison of Non-EVO Algorithms on wumpus-1
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

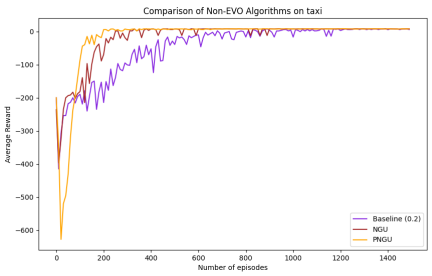

Comparison of Non-EVO Algorithms on wumpus-2
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

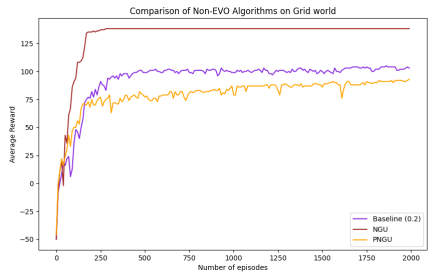

Comparison of Non-EVO Algorithms on wumpus-3
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

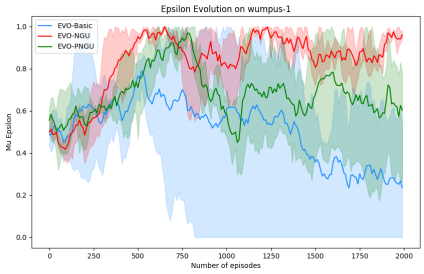

Comparison of Non-EVO Algorithms on wumpus-4
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

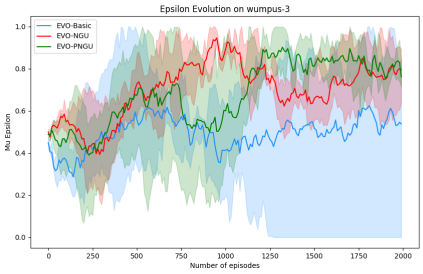

Comparison of Non-EVO Algorithms on wumpus-5
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

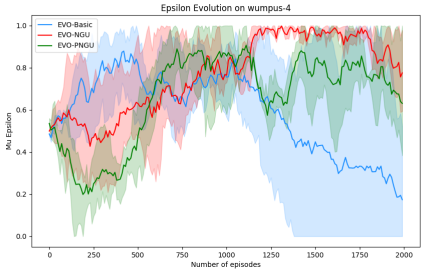

Comparison of Non-EVO Algorithms on wumpus-6
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

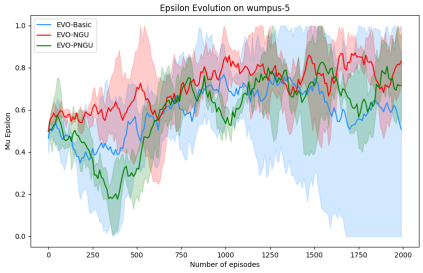

Comparison of Non-EVO Algorithms on taxi
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

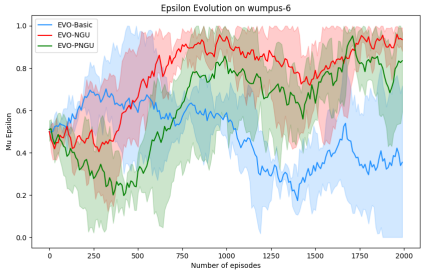

Comparison of Non-EVO Algorithms on Grid world
Baseline (0.2)
NGU
PNGU
Average Reward
Number of episodes

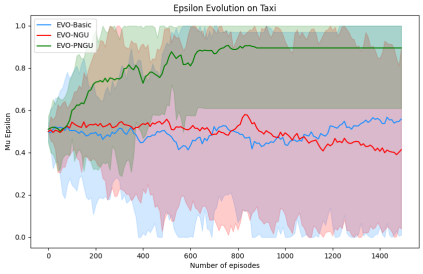

Epsilon Evolution on wumpus-1
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes

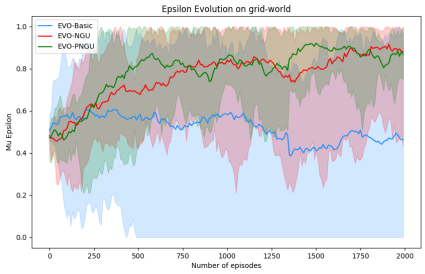

Epsilon Evolution on wumpus-3
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes


Epsilon Evolution on wumpus-4
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes


Epsilon Evolution on wumpus-5
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes


Epsilon Evolution on wumpus-6
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes


Epsilon Evolution on Taxi
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes


Epsilon Evolution on grid-world
EVO-Basic
EVO-NGU
EVO-PNGU
Mu Epsilon
Number of episodes

Appendix B

Hyperparameter Selection

These hyperparameters were mostly chosen using a grid search method. The algorithm-dependent parameters are the same across all environments and are therefore generalized in our algorithms. The environment-dependent parameters are specific to each environment, and the epsilon hyperparameter is only used in non-Evo algorithms.

B.1 Algorithm dependent hyperparameters

hyperparameters	**Value**
N_{actor}	2
k	6
β	0.3
ϵ_α	7

Table B.1: NGU hyperparameters

hyperparameters	**Value**
N_{actor}	1
k	6
β	0.3
batch size	500
replay buffer capacity	5000
replay buffer alpha	0.6
replay buffer beta	0.4
ϵ_α	8

Table B.2: PNGU hyperparameters

hyperparameters	**Value**
μ_0	0.5
σ	0.05
N_{agent}	3
α_{evo}	0.2

Table B.3: Evo Basic hyperparameters

hyperparameters	**Value**
k	6
β	0.3
ϵ_α	7
N_{agent}	3
N_{actor}	2
α_{evo}	0.2
μ_0	0.5
σ	0.05

Table B.4: Evo NGU hyperparameters

hyperparameters	Value
k	6
β	0.3
batch size	500
replay buffer capacity	5000
replay buffer alpha	0.6
replay buffer beta	0.4
ϵ_α	8
N_{agent}	3
N_{actor}	1
α_{evo}	0.2
μ_0	0.5
σ	0.05

Table B.5: Evo PNGU hyperparameters

B.2 Environment dependent hyperparameters

hyperparameters	Value
α	0.1
γ	0.99
ϵ	0.8

Table B.6: Frozen Lake hyperparameters

hyperparameters	Value
α	0.9
γ	0.99
ϵ	0.2

Table B.7: Taxi hyperparameters

hyperparameters	**Value**
α	0.9
γ	0.99
ϵ	0.2

Table B.8: Wumpus hyperparameters

hyperparameters	**Value**
α	0.9
γ	0.99
ϵ	0.2

Table B.9: Grid World hyperparameters

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**