UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Universitat de Barcelona
Facultat de Matemàtiques i Informàtica

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Master Thesis

Reverse Curriculum Hierarchical Recursive Learning

Author: **Tair Tahar**
Supervisor: **Mario Martin Muñoz**, Department of Computer Science, UPC

Master in Artificial Intelligence (MAI)

Barcelona, June 2022

Abstract

This work presents a study on Hierarchical Reinforcement Learning, where several different approaches for learning are researched, developed and tested. Specifically, the algorithm **Reverse Curriculum Vicinity Learning** (RCVL) resulted with an excellent performance in the tested environments. It is built on two level hierarchies, where the high hierarchy learns to suggest adequate subgoals to the lower level recursively, the latter learns the sequence of needed primitive actions to achieve the subgoals, and finally the ultimate goal. Currently it is designed only for discrete Reinforcement Learning environments.

Hierarchical Learning allows to break a task into several smaller sub-tasks, which results with faster learning, since smaller tasks are easier to master. Each of the levels in the hierarchy has its own "resolution" (i.e. different time scales) of the problem, while the low policy is the one to interact with the environment exclusively. In our problem the subgoals proposed by the high policy can be seen as milestones that break the big task into several shorter tasks.

The proposed algorithm integrates the concept of Reverse Curriculum Learning. Its learning begins from states around the goal, and gradually expands to more difficult tasks from further states, until mastering the whole state space. With this curricular approach, the agent is able to learn faster: first it masters the easy tasks, and then challenged with harder tasks. In the proposed algorithm the high hierarchy stores neighbours from the vicinity of each goal (collected by low hierarchy interactions) such that the goal is reachable from them with a limited number of actions. In the meantime, the low policy learns simple actions to solve the mini-trajectories from the neighbours to the goal. Then with the accumulation of knowledge of both hierarchies, the high policy learns to draw a path from the goal to the state backwards recursively, suggesting the subgoals along the way. By long-term return estimation learning, the agent is able to decide which is the best subgoal for each given pair of state and goal (or subgoal).

It is possible to make the learning even faster when allowing a preliminary phase of vicinity acquisition. In this configuration small and fast interactions of the low level with the environment are done first. Only afterwards the concurrent learning of both hierarchies is performed together. This allows a good starting point for learning during the main learning phase.

More concepts are integrated in the algorithm to accelerate the learning and to allow sample efficiency. First, the algorithm is an off-policy algorithm, that is, the experience is accumulated, and is being used later to train a policy that is different from the policy acting while the experience was acquired. Secondly, the reward system is designed to exploit the maximum information when rolling-out the collected experience so that all possible ordered combinations are stored with a non-sparse reward. Finally, the algorithm uses Hindsight Experience Relabelling, allowing exploitation of the accumulated experience in a more efficient way.

The proposed algorithm has shown outperformance over the State of the Art algorithms: DDQN [1], and DDQN combined with HER [2] in more complex tested environment. It has reached to a success rate of above 97%, while avoiding unfeasible subgoals suggestion, and constructing optimal paths. Finally, it has shown to be robust to most of hyperparameters changes.

Keywords: AI · Machine Learning · Reinforcement Learning · Hierarchical Learning · Curriculum Learning · Actor Critic · SAC · DDQN · HER · Q-learning · UVFA

Acknowledgements

I would like to use this opportunity to give my gratitude to important people who's presence was essential along this journey:

To my amazing parents, with their endless unconditional giving and caring, which is impossible to describe with words. Together with the rest of the family, you have given me the strength and calmness I needed.

To my wonderful partner, Maria, for being my best friend, my family, my confidant. For being supportive and encouraging, and for making Barcelona a home.

To my great friends from MAI, for the reciprocal teaching with patience and laughter, and for sharing with me beautiful moments in Catalunya.

To my dearest long-life friends from Israel, that are always there for me. to ask, to listen, to be. It would have not be possible without them.

To Varda, who always knew why.

To the Barcelona Supercomputing Centre, for facilitating the computational challenges with kindness and efficiency.

Finally and mostly, a greatest gratitude to Mario my dear supervisor, who has ignited my passion for Reinforcement Learning, and gave me generous guidance through this quest. Thank you for incredibly interesting conversations and fruitful shared thinking.

Contents

Acronyms

AC Actor Critic. 14, 15

CL Curriculum Learning. 24

DDPG Double-Deep Q-Networks. 20

DDQN Double Deep Q-Network. 31, 58, 60

DQN Deep Q-Netowrk. 12

DRL Deep Reinforcement Learning. 20

ER Experience Replay. 11, 12, 18, 32–34, 36, 38, 44

FA Function Approximator. 11, 12

HER Hindsight Experience Replay. 20, 21, 29, 31, 34–36, 39, 44, 58, 60

HL Hierarchical Learning. 23, 24

MC Monte Carlo. 10, 11, 14

MDP Markov Decision Process. 18, 27

NN Nueral Network. 12

PI Policy Iteration. 6, 9, 10, 12

RL Reinforcement Learning. 1–5, 7–9, 20, 24, 29

SAC Soft Actor Critic. 16, 19, 32, 33, 65

TD Time Differences. 11

UMDP Universal Markov Decision Process. 31, 32

UVFA Universal Value Function Approximation. 18–20, 33

VI Value Iteration. 6, 9, 10

Chapter 1

Introduction

This projects presents our two new developed algorithms `Reverse Curriculum Recursive Learning` and `Reverse Curriculum Vicinity Learning`, that were designed for solving complex `Reinforcement Learning` environments tasks with both high sample efficiency and high learning efficacy. We make use of some concepts as off-line learning, hierarchical learning, hindsight experience relabeling, and curriculum learning.

1.1 Motivation

Reinforcement Learning (RL) is a field in Machine Learning, where the learning is done by an agent that explores an environment and tries to solve a task. The way it learns mimics human learning: **Trial and Error**, by interacting with the environment, accumulating experience that includes rewards on the actions it does, when it is found in a particular state. The agent aims to maximise its reward in the long term, meaning it needs to learn a behaviour that is adequate to the environment and that brings it closer to solving the task. The research in this field has different areas of focus, from the reward function, through agent architecture, experience accumulation, the balance between exploration of new revealed states and exploitation of the existing knowledge and more.

The immediate reward the agent receives from the environment for performing an action, can be binary (0 or 1), positive or negative, or sparse, meaning it does not receive anything until a trial is ended, or a combination of those. Most of the environments in real world problems have binary and sparse rewards: if the agent achieved the goal it gets 1 in the end of the trial, otherwise 0 (or 0 and -1 instead, respectively). That makes it very difficult for the agent to learn, and it means that many trials end up with no contribution for the learning - low learning signal. If the agent fails there is no indication if it is because it was far from achieving the goal, or maybe only a few steps were missing for success. That affects sample efficiency, since experience does not necessarily help to learn. That is one reason for the limited applicability of RL algorithms.

In addition to the insufficiently indicative reward, the applicability of `RL` algorithm on real-world problems is limited since it requires both `RL` expertise and domain expertise. Furthermore, there are cases where the admissible behaviour is not known.

However, the field is experiencing a significant progress in RL that brought to the developments of more complex agent architectures, and to the need of parallelism in both reward estimation and policy learning. Trying to tackle the inefficiencies of the process we aspire to propose an algorithm that combines several paradigms which form together an efficient and generalizable algorithm.

First, we use **Hierarchical Learning** (HL), that allows breaking down one big task to several small tasks, such that each hierarchy learns tasks in different resolution, where the combination of the hierarchies together is able to solve the big task more efficiently. The fact that each of the hierarchies has only part of the big tasks, allows the agents to learn much faster complex tasks. The lower hierarchy is the one that performs interactions with the environment. An example for that would be if we want a robot to pick up an object and locate it in another location. The high level steps of the process would be in that case: to move the robotic arm to a current object location, grasp the object, move to the desired location and finally leave the object. The low level tasks would be the series of primitive actions that result with each of the high level steps. For instance, to pick up an object, the robot needs to learn how to close its arm around the object such that it will not fall when moving it. In order to reach a specific location, it needs to try to advance from any state to the desired, via exploring new states and exploiting the accumulated knowledge.

Second, the experience the agent gains interacting with the environment, is stored in the **Experience Replay** (ER) for later networks update, formally **Off-policy learning**, meaning the agent learns from other behaviours (since it keeps changing its own behaviour), allowing sample efficiency. To make use of the experience even more, **Hindsight Experience Replay** (HER) helps a lot. It suggests **relabeling** existing experiences, even if they were not helpful for the current goal, they might be useful for the case that the goal is the actual state the agent ended up reaching.

The third element is the **Curriculum Learning** where the subgoals that are suggested to the agent come in a certain order, mimicking the curricular process: in the beginning easy goals are suggested, and as the agent becomes more trained, and masters its current goals, it is possible to challenge it with harder goals gradually. Different ways of creating an automated curriculum are investigated and developed. The paradigm we adopted is the **Reverse Curriculum**, where we start learning from easy starting states (close to the goal) and gradually make the problem harder, learning to reach the goal from distant states.

The solution is formed by several **Neural Networks** (NN) that act like function approximators for the long-term return and the behaviour of the agent. That is an important on-going developing sub-field of **Deep Reinforcement Learning** that presents the State of the Art algorithms currently studied.

Finally, the purpose of this work is to suggest an algorithm that combines the elements above, and to test it in possible environments.

1.2 Contribution

This work aims to present a new **Hierarchical Reinforcement Learning** algorithm, that allows an agent to solve relatively complex problems, while demonstrating sample efficiency, and avoiding unacceptable states. The solution is composed by two levels hierarchy, where the higher suggests subgoals or promising starting states to the lower level recursively expressing **Reverse Curriculum** approach. The reward is shaped to be non-sparse, which makes the learning more efficient.

With the mentioned ideas, and the combination of **Hindsight Experience Relabelling**, the lower level is able to master the short trajectories from states in the vicinity of a goal to the goal, and in a recursive way, it knows to reach from any starting state to any goal state in the given environment, passing through the milestone given by the high level policy. The high level policy proficiency in high level tasks evolves as the lower level performs interaction with the environment. It can be seen like the high policy ”samples” knowledge from the low policy with a “sample rate” of the low policy horizon. Low policy horizon is the top limit number of primitive steps we set for each mini-trajectory performed. The algorithm as a whole challenges currently existing State of the Art algorithms, presenting very impressive results for a discrete space.

1.3 Document Overview

This report suggests three main RL algorithms that combine several fields of research. Chapter 2 gives the background and goes through the formulation of the given problem. Chapter 3 presents some State of the Art papers and ideas in the same area of research, while some of the ideas presented are integrated in suggested algorithms. Chapter 4 describes the algorithms in detail, while chapter 5 presents the most relevant experiments. Lastly, chapter 6 presents some conclusions and possible future research and development.

Chapter 2

Background: Reinforcement Learning

This chapter presents the background of Reinforcement Learning and the formulation of the problem, while emphasising the concepts and paradigms that are used in the proposed solutions. Unless cited otherwise, the information presented in this chapter is summarised from Sutton et al. introductory book for RL [3] and Professor Martin Course Notes [4].

2.1 Initial Formulation

Reinforcement Learning (RL) is a field of learning in `Machine Learning`, where the learned skill is a behaviour given a specific environment and tasks, that are referred to as goals. The idea is that an agent is learning how to achieve the goals, getting rewards from the environment on the actions it chooses to execute: **trial and error** just like a human being. A positive reward encourages the agent to choose the selected action in the future, while a negative (penalty) should do the opposite. Eventually the expectation is that the agent will learn how to achieve its goals in an effective and efficient way, converging to the optimal solution (optimal behaviour).

The main elements of a RL are [5]:

- `Environment` - the world where the agent executes its tasks.
- `State` - current agent's status.
- `Reward` - the payment the agent receives after executing an action in the environment given its current state.
- `Policy` - the behaviour of the agent, which can be seen as a mapping between current state and goal, to an action in the action set. It can be either deterministic or stochastic.
- `Value function` - the accumulated expected future reward when executing an action given the current state. When speaking on `V-value` function, then the actions selected along the way are according to the agent policy, while `Q-value` function, is where current action is not necessarily the current policy choice, but then the rest of the action are aligned with the policy.

The general framework of RL, can be represented by figure 2.1 and it can be formalised by `Markov Decision Process (MDP)`, as the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, P >$. The current state of the agent is assigned as s, which is one state from the set of possible states $\mathcal{S}$. By executing action a out of the set of possible actions $\mathcal{A}$, the agent receives a reward $r \in \mathcal{R}$, while the state of the agent is updated to be $s' \in \mathcal{S}$. When the environment is not fully observable, then instead of state we have only `observation` o, which might not contain the complete knowledge about the real state. $\mathcal{R}(s_t, a_t, s_{t+1})$ is the reward function, and $P(s_t, a_t, s_{t+1})$ is the transition probability function in time step t.

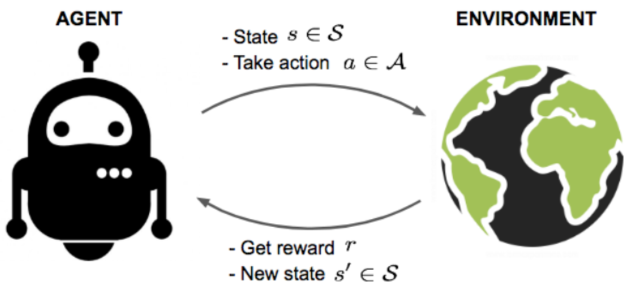

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 2.1: *RL* Framework [6]

The objective of the process is for the agent to learn an optimal behaviour (policy), which maximises its overall long term return. The reward function $\mathcal{R}$ determines the immediate reward r_t at each step of the learning $r_t = \mathcal{R}(\mathcal{S}_t)$:

$$\mathcal{R}(s, a) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a] \tag{2.1}$$

Which can also be expressed in the following way:

$$\mathcal{R}(s, a) = \sum_{s'} P(s, a, s') \cdot \mathcal{R}(s') \tag{2.2}$$

The **long-term reward** in the case of a finite horizon $\mathcal{H}$ can be express in the following way:

$$\mathcal{R}_t = r_{t+1} + r_{t+2} + r_{t+3} + ... = \sum_{k=0}^{H} r_{t+1+k} \tag{2.3}$$

For the infinite horizon case, we say the the summation continues until infinity (H $\rightarrow \infty$). A discount factor $\gamma \in (0, 1]$ is usually introduced when calculating the **long-term reward**, which results with the following expression for the infinite horizon case:

$$\mathrm{R}_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} ... = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \tag{2.4}$$

When the value of γ equals one, we get a far-sighted evaluation, while when it is close to zero, the significance of further steps decreases greatly.

2.2 Value Function

The `Value Function` is a prediction of future accumulated reward from a particular state, which depends on the agent's policy. The idea is to have the cached knowledge in a single function $V^\pi(s)$ that represents the utility of any state s in achieving the agent's overall goal or reward function. This knowledge allows the agent to immediately assess and compare the utility of states and/or actions [7]. The `state-value function` $V^\pi(s)$ is defined as the expected return, when the agents begins at state s and follows the policy π:

$$V^\pi(s) = \mathop{\mathbb{E}}_\pi [R_t | S_t = s] = \mathop{\mathbb{E}}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s \right\} \tag{2.5}$$

The `action-value function` $Q^\pi(s,a)$ is the future expected return when the agents begins at state s, executes action a and the follows then policy π:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_\pi \left[R_t \middle| s_t = s, a_t = a \right] = \mathop{\mathbb{E}}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{2.6}$$

The **Bellman expectation equation** decomposes the `value function` and `action-value function` using the recurrence relation in following way:

$$V^\pi(s) = \mathop{\mathbb{E}}_\pi [R_t | s_t = s] = \mathop{\mathbb{E}}_\pi \left[r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s \right] \tag{2.7}$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_\pi [R_t | s_t = s, a_t = a] = \mathop{\mathbb{E}}_\pi \left[r_{t+1} + \gamma V^\pi(s_{t+1}) \middle| s_t = s, a_t = a \right] \tag{2.8}$$

We can notice that $Q^\pi(s, \pi(s)) = V^\pi(s)$, which implies that the above can be also expressed in recursively with Q state-action value:

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_\pi \left[r_{t+1} + \gamma Q^\pi \left(s_{t+1}, \pi(s_{t+1}) \right) \middle| s_t = s, a_t = a \right] \tag{2.9}$$

The `optimal policy` is a policy that means any action selection that is not following the policy will result in lower long-term return. The learning of the optimal policy can be achieved by either iterating the policy (PI) adding some exploration or iterating the value functions (VI), while some algorithms combine them both. In order to converge to the optimal policy, the agent faces the trade-off between exploration of new unknown state, to exploitation of the information it already obtained from the environment in past experience.

It is possible to improve policy π if and only if there exist a state s in the state space $\mathcal{S}$ and an action a in the action space $\mathcal{A}$, such that $Q^\pi(s,a) > Q^\pi(a, \pi(s))$. If a policy π is improvable, it means it is not optimal when choosing action a from state s.

The optimal policy can be expressed in the following way based on value functions and transition function, showing is it a **greedy policy**:

$$\pi(s) = \operatorname*{argmax}_{a \in \mathrm{A}} Q^\pi(s,a) = \operatorname*{argmax}_{a \in \mathrm{A}} \sum_{s'} P^a_{ss'} [R(s') + \gamma V^\pi(s')] \tag{2.10}$$

If the policy is the greedy policy that means that the value function is maximised:

$$V^{\pi}(s) = \max_{a \in \mathrm{A}} Q^{\pi}(s,a) = \max_{a \in \mathrm{A}} \sum_{s'} P^{a}_{ss'}[R(s') + \gamma V^{\pi}(s')] \tag{2.11}$$

2.3 Stochastic Vs. Deterministic Policy

A policy can be either **stochastic** or **deterministic**. **Deterministic policy** would like:

$$a = \pi_{\theta}(s) \tag{2.12}$$

meaning that in state s, the agent will always choose action a. **Stochastic policy** would be expressed like:

$$P(a|s) = \pi_{\theta}(a|s) \tag{2.13}$$

meaning that the probability for the agent to choose action a from state s is according to the policy. Stochastic policies can be either **categorical policies** in discrete action space, or **Gaussian policies** for continuous action space. The good thing with stochastic policies is that they are smoother than greedy policies, so it is possible to use the gradients.

For training stochastic policies we can sample actions from the policy, and we can compute the log likelihood of particular action $log\pi_{\theta}(a|s)$.

2.4 Exploration Vs. Exploitation

A `Greedy Policy` is the optimal one. That is the behaviour that has the highest possible benefit given a particular environment. It can not be further improved. A greedy action is the one that is selected by the policy. Under the assumption that the policy converges to the optimal one, a greedy action is simply the one that is suggested by the current policy. However, we know that the policy is not always optimal, so its selection of actions should be reconsidered during training. Mostly in the beginning of the training, where we start with a random policy, it is hard to believe that the action selected by the policy is the best one.

Knowing the agent collects its experience, and in the meanwhile aims to improve its policy, we face the RL famous trade-off of exploration and exploitation. In order to get to know as much as possible about the environment, to be able to choose wisely, and converge to the optimal policy, we need to have the experience of the agent as wide as possible. That means we need to explore a lot, and let the agent not always go with its current policy, so new states are visited, and the knowledge on the environment gets more complete. On the other hand, we want to maximise the long-term expected return. We have the policy that is learnt, and that is expected to get closer and closer to optimal behaviour, and assumed to benefit the agent with higher rewards. That makes it appealing to use the existing policy, knowing that following it, the long-term return should be relatively high (higher as much as the training proceeds). With that, if we keep only following the current policy, we might get stuck in a local minimum, with suboptimal behaviour, because we haven't got all the experience needed to do the best actions.

A useful compromise between those two is the **ϵ-greedy (Epsilon Greedy)** exploration. That method allows the agent to explore with probability of ϵ. Explore means to choose an action at random. The agent chooses the action according to the policy only with probability of $1 - \epsilon$. When ϵ is high (close to 1) then we are more likely to explore, and the opposite when it is small. ϵ can be viewed as a hyperparameter, where the problem needs to be solved and the agent's architecture should be considered for a wise choice. It is frequent to use a decaying epsilon, which means that in the beginning of the learning we explore more, and as we proceed we rely more on the policy, as the policy is closer to the optimal one, less random actions for exploration.

There exist some other popular exploration methods such as `Softmax Exploration`, `Gibb's exploration`, that adjust the exploration with more specific characteristics to bias the exploration to more promising actions, or other possible motivations.

2.5 On-policy Learning Vs. Off-policy Learning

`On-policy` methods try to evaluate/improve the policy that is used for action selection, while `Off-policy` methods try to evaluate/improve a policy that is different from the one that is used for the data generation. In `on-policy` the agent grasps the optimal policy and uses the same policy to act and learn from it as the action is executed, right after the episode completion.

`On-policy` training should be used when the value of the current exploring agent needs to be optimised. `Off-policy` training can be cost-effective when the deployment of the algorithm is in the real world. The updated policy is different from the policy used for experience accumulation. With that said, for the `off-policy` case, the evaluation becomes more challenging due to high randomness.

2.6 Model free algorithms

RL problems can be solved using **model-based** methods or model-free methods. Model-based methods allow inference on the environment [8]. Models are used for planning, deciding on actions by considering possible future situations. A model predicts the next state and the reward. In **model-free** solutions, we can not infer on the environment, and the information about the environment is captured only by the interaction of the agent with it: being in a particular state s, perform an action a, and observe the reward r and the next state s'. Changing the policy of value function from a specific state of a model-free agent, the agent must move to that state and act from it, possibly many times. Model-free algorithms are able to generalise to more environments.

In figure 2.2 it is possible to see the distribution to the two main groups, through the methods used and eventually specific algorithms that implement these ideas. The algorithms that will be presented in this work are model-free, which allows generalization to other possible environments.

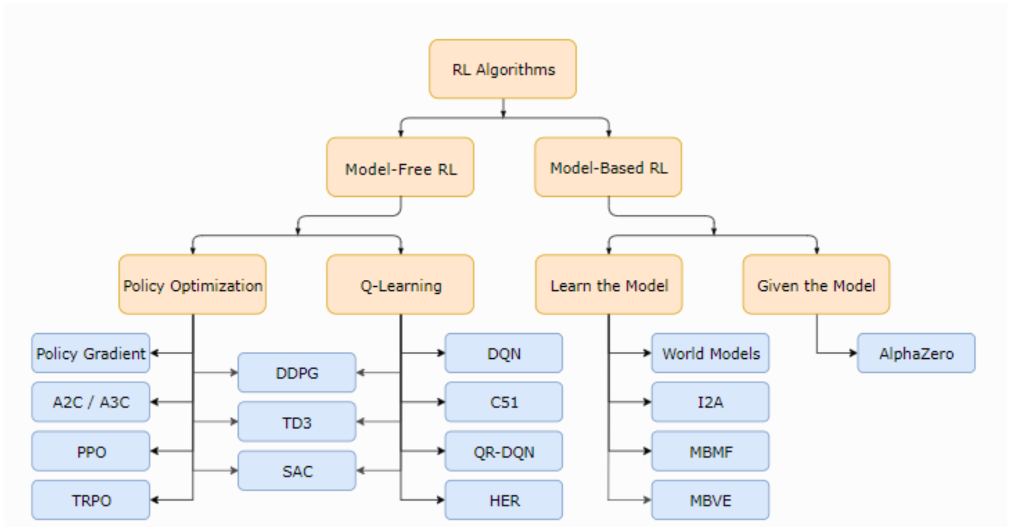


Figure 2.2: Taxonomy of RL algorithms[9].

In **model free** algorithms the agent gets to know more and more about the environment, as it collects experiences, while interacting with the environment. The experience is collected while the agent is going through different **episodes** or **trials**. Each episode has a starting state and a goal state. The observation of a sequence of states, actions and reward during the episode reveals information regarding the environment. Based on those experiences, it is possible to estimate the long term return from each state by averaging the collected rewards after these states in different episodes.

In the following sections several important components of model free algorithms will be presented, as well as some useful methods we will use in this project. We will start by explaining VI and PI methods, and then we will see how those are merged into one algorithm with the two components.

2.6.1 Value Iteration

One major drawback of PI (that will be elaborated in more details in section 2.6.2), is that each iteration involves policy iteration, which is another iterative process over the state space. Sutton et al. show in their book [3] that there is no need to iterate the value function more than a few times, to lean on it for policy evaluation, without losing the promise for convergence of the algorithm. An important special case is where value iteration is truncated after exactly one iteration, it is called `Value Iteration`, and it combines policy improvement and truncated policy evaluation steps. In practice, VI can be seen as turning `Bellman's` optimality equation into an update rule. Similar to PI this process should continue an infinite amount of times until convergence to the optimal value, however when the updates are small enough (below some threshold) the process is stopped in practice.

2.6.1.1 Monte Carlo Methods

The `Monte Carlo` methods assume episodic tasks, meaning every episode terminates, with no dependency on the action selection. In those methods we average the complete return from the state until the end of the episode. The process is involved with computing the value functions V^{π} and Q^{π} for a fixed arbitrary policy π, then policy improvement, and finally generalised policy iteration. This idea is used also in `Dynamic Programming (DP)`, which is a `model-based` approach.

In its simplest version, `MC` takes long term rewards collected from a state in different episodes and averages over them. For this case, it is necessary to complete all the episodes we want to base on in the calculation. In the version of `Incremental MC` it is possible to update the value function after every episode, following the update:

$$V_n(s_t) = V_{n-1}(s_t) + \frac{1}{n}(R_t - V_{n-1}(s_t)) \tag{2.14}$$

Where n is the number of visits to each state, which we should store. Instead of using that coefficient of $\frac{1}{n}$ usually a constant parameter α is used:

$$V(s_t) = V(s_t) + \alpha(R_t - V(s_t)) \tag{2.15}$$

It can be shown that:

$$V_n(s) = \alpha \sum_{i=0}^{n-1} \left[(1-\alpha)^i R_{n-1} + (1-\alpha)^n R_0\right] \tag{2.16}$$

With this parameter α it is possible to control the proportion of old long-term return forgetting. It is useful since as the training proceeds, the policy is updated, so the new created experiences are more updated with it, and older ones are less important. Moreover, it is frequent to use a decaying alpha, meaning forgetting more.

Observing equation 2.15, but bearing in mind the exploration - exploitation trade-off that is brought under section 2.4, the MC incremental update becomes:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R_t - Q(s,a)) \tag{2.17}$$

MC is an `on-policy` method (further explanation on on-policy learning is under section 2.5).

2.6.1.2 Temporal Difference (TD) Methods: Q-Learning

Another value iteration method that uses an idea that is similar to MC, but using `Bellman's` equation in the place of R_t:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r_{t+1} + \gamma Q(s', \pi(s')) - Q(s,a) \right] \tag{2.18}$$

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \left[r_{t+1} + \gamma Q(s', \pi(s')) \right] \tag{2.19}$$

This way of using `Q-Value` estimations of the next state to update the current one is called `bootstrapping`, and it allows the value to update every state and not only when the episode ends. That is a significant advantage of `Q-Learning` over MC. TD is an `off-policy` method (further explanation on off-policy learning is under section 2.5).

2.6.1.3 Deep Q-Netowrk

`Q-Learning` that is performed via MC and TD is based on the tabular case of Q-function, which becomes very complex as the dimensionality grows. That is what gives the importance to **Function approximators** (FA). Its major advantages are that it can both generalise to unseen states, and in the meantime enabling to approximate a Q-value table without a dependency on the state/action dimensionality.

The way to converge to optimality would be applying `gradient descent` where the loss function is the difference between the estimation and the actual real values from the collected experience, in a supervised manner. When the real value is unknown then bootstrapping is used.

Having this approach alongside with online incremental learning, the experience samples are consequent and are not independent and identically distributed (i.i.d). This problem is resolved with `batch gradient descent`, which means that we sample randomly from past experiences. This of course can be done only with off-policy learning, using a `Replay Buffer` or `Experience Replay ER`. This buffer accumulates the collected experiences, and then when it is needed to update the approximator, a sample from this buffer is used. Experience is stored in the ER with the pattern of $< s, a, r, s' >$: state,action, reward and next state. In this way the samples are expected to be closer to be i.i.d and the learning gradient is more stable since it relies on computation over a batch of samples and not only a single one experience.

Another problem that rises is the **moving target** problem. We calculate the loss based on the target that keeps on changing. The solution that is suggested is to handle two sets of approximators, θ for `Q-Value`, and the other, θ' for `target Q-approximator`. The target ones are only updated when an episode is finished, meaning it does not change during supervised regression. That solves the problem of the moving target to a large extent.

In addition with ER, having the policy changing over the time, but continuing sampling from old experiences for approximator updates, might result in inappropriate updates for irrelevancy purposes. That can be solved with either removing old samples from ER, or limit its size with `FIFO` (first in, first out) method cleaning. `FIFO` means that the first one to get into the ER is the first to also get erased from the memory.

DQN algorithm is an important breakthrough [10] in `Q-learning` approximation development. It is a FA in the form of a deep Neural Network (NN) along with incremental learning mode. It uses a NN, parametrized by θ for $Q_\theta(s,a)$ approximation, while for loss calculation, the ground truth that is used is bootstrapping for each experience (s,a,r,s'):

$$Q_\theta(s,a) = r + \gamma \max_{a'} Q_{\theta'}(s',a') \tag{2.20}$$

The error can be then expressed by:

$$error = Q_\theta(s_t,a_t) - r_{t+1} + \gamma \max_{a'} Q_{\theta'}(s_{t+1},a') \tag{2.21}$$

Based on the value FA the policy can be evaluated and improved.

DQN has revealed a clear outperforming over the former existing methods, however one significant drawback is that it can be optimistic when estimating the `Q-value`. That happens given a random initialization of the networks. Observing equation 2.21, we notice that in the case where all of actions have a zero reward, the estimation should be exactly r, however with the random initialization it might get a positive result (this is a maximum operator). This overestimation propagates to other states.

2.6.1.4 Double Deep Q-Learning (DDQN)

`Double Deep Q-Learning` [1] tackles the overestimation problem presented in the former section, using the two existing NNs for `Q-value` estimations that are parametrized by θ and θ'. The NN $Q_{\theta'}$ is udes for action selection while Q_θ is used for action evaluation.

The equation for the error in 2.21 then becomes the following:

$$error = Q_\theta(s_t,a_t) - r_{t+1} + \gamma \max_{a'} Q_\theta(s_{t+1}, \operatorname*{argmax}_{a'} Q_{\theta'}(s_{t+1},a')) \tag{2.22}$$

The resulting algorithm is shown in Algorithm 1. In this algorithm we minimise square error between Q_θ and Q^*, while $Q_{\theta'}$ softly copies the parameters of Q_θ, using `Polyak averaging` with τ, the averaging rate [11].

2.6.2 Policy gradient algorithms

In this family of algorithms there is an explicit representation of the policy $\pi_\theta(a|s)$. The parameters θ are learnt via optimising the objective function (loss). The general idea of PI is to initialise the policy randomly and then go through the iterative process where the action selected by the policy is updated in case it increases the long-term return. The optimal policy is expressed in 2.10.

Algorithm 1 DDQN: Deep Double Q-Learning (Hasslet et al., 2015) [1]

```
1: Initialize primary network $Q_\theta$, target network $Q_{\theta'}$, replay buffer $\mathbb{D}$, $\tau << 1$
2: for each iteration do
3:     for each environment step do
4:         Observe current state $s_t$ and select action $a_t \sim \pi(s_t)$
5:         Execute and observe next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
6:         Store experience $(s_t, a_t, r_t, s_{t+1})$ in $\mathbb{D}$
7:     end for
8:     for each update step do
9:         Sample experience batch $\sim \mathbb{D}$
10:        Compute target $Q$ value: $Q^*(s,a) \approx r_t + \gamma Q_\theta(s_{t+1}, \text{argmax}_{a'} Q_{\theta'}(s_{t+1}, a'))$
11:        Perform batch gradient descent step on the error $(Q^*(s,a) - Q_\theta(s,a))^2$
12:        Update target network parameters: $\theta' \leftarrow \tau \cdot \theta + (1-\tau) \cdot \theta'$
13:    end for
14: end for
```

In the end of every iteration of training, or every several of them, the policy that is learnt should be evaluated, which means its `value-function` should be calculated. In theory the process should continue infinite times for convergence to the optimal policy. In practice, the iterations continue until the updates in the policy become very minor (below some threshold).

Policy iteration has more promising probability for convergence than `Q-value` iteration. It is effective in high dimensional or continuous action spaces, and it can learn stochastic policies. However, it tends to converge in a sub-optimal solution (locally optimal), and it also tends to have a lot of variance and to sampling inefficiency.

There are several approaches for policy iteration such as `Cross Entropy Method` and some `Genetic Algorithms`. We will elaborate more on `Policy Gradient` since it is the approach we implement in this project.

The idea is to perform `gradient ascent` to find the policy that results with the **maximum long term return**, the value function. We define the the long-term reward from a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T, s_T)$ as the sum of the reward accumulated during the trajectory, which can also be discounted with γ:

$$R(\tau) = \sum_{t=1}^{T} r(s_t) \tag{2.23}$$

We then note $P(\tau|\theta)$ as the probability of the path τ following policy π_θ. The policy value $J(\theta)$ can be expressed as in equation 2.24. We aim to maximise this term with respect to θ, which implies analytically compute $\nabla P(\tau|\theta)$. Assuming the policy is differentiable when it is not zero, we can use the log trick that allows us to replace this gradient with the derivative of its logarithm as shown in equation 2.25. Then we can derive $\nabla_\theta log P(\tau|\theta)$ using the stochastic policy to eventually get the gradient expression brought in 2.26 (H is the horizon) for a sample of m experiences.

$$J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[R(\tau)] = \sum_{\tau} P(\tau|\theta)R(\tau) \tag{2.24}$$

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau|\theta)R(\tau)\nabla_\theta logP(\tau|\theta) \tag{2.25}$$

$$\nabla_\theta J(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} R(\tau)\sum_{i=0}^{H-1}\nabla_\theta log\pi_\theta(a_i|s_i) \tag{2.26}$$

Second approach that aspire to optimise the expected return over state (2.27), having d^{π_θ} the probability of being in each state according to the policy (expected number of time steps the agent is in s normalised by the number trial time steps). Third and last, taking into consideration one immediate reward as brought in 2.28.

$$J(\theta) = \sum_{s} d^{\pi_\theta}V^{\pi_\theta}(s) = \sum_{s} d^{\pi_\theta}\sum_{a}\pi_\theta(a|s)Q(s,a) \tag{2.27}$$

$$J(\theta) = \sum_{s} d^{\pi_\theta}V^{\pi_\theta}(s) = \sum_{s} d^{\pi_\theta}\sum_{a}\pi_\theta(a|s)r(s,a) \tag{2.28}$$

Policy Gradient theorem expresses the equation for the objective function gradient, as appear in 2.29, for each of the approaches proposed (2.24, 2.27, 2.28).

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^{\pi_\theta}(s|a)] \tag{2.29}$$

The classic algorithm to mention under policy gradient is `Reinforce`. It is a simple policy iteration where we generate an episode following the policy π_θ, then rolling it out we calculate the long-term return from each time step R_t, and finally updating the parameters θ with a rate of α and according to: $\nabla_\theta log\pi_\theta(a|s)R_t$ based on 2.29. This process is being iterated until convergence up to some threshold.

One problem in `Reinforce` is that it has a lot of variance, which affect the convergence and solution. To overcome this, a baseline, which is an estimation of $V^{\pi_\theta}(s_t)$ is introduced for the calculation of R_t. This approach is called `Actor Critic` algorithm.

2.6.3 Actor Critic Algorithms (AC)

This is a family of algorithms that combines both `Policy Iteration` and `Value Iteration` concurrently. AC can be performed as on-policy (MC) or off-policy. It uses the estimation of $V^{\pi_\theta}(s_t)$ as a baseline in the parameters update and explained in the former section.

It has two main components: An `Actor` that implements the current policy, and a `Critic` that evaluates the current policy, and is being used for training. It holds two sets of parameters, one for each one of the components. For the `Critic` approximation is it possible to apply the explained under section 2.6.1, including bootstrapping and more. The `Actor` ascends with the gradient, directed by the `Critic`.

The diagram of one step AC can be visualised in figure 2.3. TD (time difference) is for calculating the error, by the `Actor` and the `Critic` for the update. This algorithm has a lot of variations: λ time steps error, `Trust Region Policy Optimization (TRPO)`, which optimises action with respect to existing policy by proximal steps, `Natural Actor Critic`, which assures the direction of ascending with the gradient is more promising towards optimality, and more.

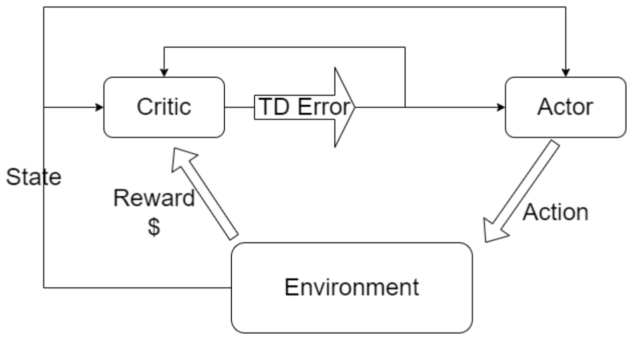


Figure 2.3: Actor Critic diagram

A famous AC algorithm is `DDPG - Deep Deterministic Policy Gradient` [12] from 2016 by Lillicrap et al., which implements an extension of `Q-Learning` (off-policy) for the continuous case. The difference from other value function approximators is that the policy is deterministic, and a gaussian is added for exploration purposes.

`TD3 (Twin Delayed DDPG)` [13] by Fujimoto et al. that was published in 2018 presented several improvements over `DDPG`. First, The actions are clipped to avoid out of the range situations. Second, a pessimism is integrated in the `Double Q-Learning`, using a **twin** Q-netowrk and take the worse for network updates. Lastly, it suggested a **delayed policy updates**, that implies more frequent `Critic` updates than `Actor` updates.

2.6.3.1 SAC (Soft Actor Critic)

`SAC` is an off-policy entropy regularised algorithm [14] [15]. It implements a stochastic policy. The entropy imposes exploration, meaning there is no need for adding noise, as done in previously presented algorithms. It makes use of 7 networks as approximators:

- Four `Q-value` networks like `TD3`: two networks to handle overestimation with target network for each of them. Those are parametrized with θ_1, $\theta_{1,targ}$, θ_2, $\theta_{2,targ}$ respectively.
- Two `V-value` networks that are parametrized by ψ, ψ_{targ} respectively.
- One `Actor` network, parametrized by ϕ.

Given the entropy that is presented in equation 2.30, the optimal policy will be 2.31, having α as a regularizer between the entropy and the reward. Normally α decays over the learning process and is set to zero in test time.

$$\mathcal{H}(\pi(\cdot|s)) = \underset{a\sim\pi(s)}{\mathbb{E}}\left[-log\pi(a|s)\right] \tag{2.30}$$

$$\pi^* = \underset{\pi}{\text{argmax}} \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s))\right)\right] \tag{2.31}$$

Let us then propagate the entropy to the expressions we used in SAC as described in 2.32 and 2.33. Those allow us to update `Bellman`'s operator to soft `Bellman`'s operator presented in 2.34 and 2.35.

$$V^\pi(s) = \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right)\middle| s_0 = s\right] \tag{2.32}$$

$$Q^\pi(s,a) = \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty}\gamma^t R(s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t\mathcal{H}(\pi(\cdot|s_t))\middle| s_0 = s, a_0 = a\right] \tag{2.33}$$

$$V^\pi(s) = \underset{\tau\sim\pi}{\mathbb{E}}\left[Q^\pi(s,a) + \alpha\mathcal{H}(\pi(\cdot|s_t))\middle| s_0 = s\right] \tag{2.34}$$

$$Q^\pi(s,a) = \underset{s'\sim P, a'\sim\pi}{\mathbb{E}}\left[R(s') + \gamma\left(Q^\pi(s',a') + \alpha\mathcal{H}(\pi(\cdot|s'))\right)\right] = \underset{s'\sim P}{\mathbb{E}}\left[R(s,a,s') + \gamma V^\pi(s')\right] \tag{2.35}$$

Back to the networks mentioned, we can now define the losses for them. The loss of each Q-value approximator is the one that appears in 2.36, while for the v-value approximator loss is as in 2.37.

The objective function for the `Actor` network is defined to minimise the `Kullback Leibler divergence` between the current policy and the Softmax policy derived from the Q-values, which makes the policy to prefer actions with high Q-values. The result expression that we would like to optimise appears in 2.38.

$$L(\theta_i, D) = \underset{(s,a,r,s',d)\sim D}{\mathbb{E}}\left[\left(Q_{\theta_i}(s,a) - (r + \gamma V_{\psi_{targ}}(s'))\right)^2\right] \tag{2.36}$$

$$L(\psi, D) = \underset{s\sim D, a\sim\pi_\phi}{\mathbb{E}}\left(V_\psi(s) - \left(\min_{i=1,2} Q_{\theta_i}(s,a) - \alpha log\pi_\phi(a|s)\right)\right)^2 \tag{2.37}$$

$$\underset{a\sim\pi_\phi}{\mathbb{E}}\left(Q^{\pi_\phi}(s,a) - \alpha log\pi_\phi(a|s)\right) \tag{2.38}$$

After obtaining the objective functions, we need to derive them for updating. The problem is that the gradients in that case are not that straightforward. The expectation depends on ϕ, the parameter we want to derive with respect to. To tackle this we define π to be a gaussian by adding noise to the action and then apply the **reparameterization trick**. We can write the actions then as appears in 2.39, where μ and σ are not stochastic. Then the expectation will not be dependent on ϕ anymore, but on the ξ sampled from a normal distribution. We, of course, need to update all of the actions accordingly. Then we are able to compute the gradient with respect to ϕ. The expectation from 2.38 can be rewritten as in 2.40, allowing optimization according to 2.41. The high level description of the algorithm is presented in algorithm 2.

$$\tilde{a}_\phi(s,\xi) = \tanh\left(\mu_\phi(s) + \sigma_\phi(s) \odot \xi\right), \xi \sim \mathcal{N}(0,1) \tag{2.39}$$

$$\mathop{\mathbb{E}}_{\xi\sim\mathcal{N}} \left[Q^{\pi_\phi}(s, \tilde{a}_\phi(s,\xi)) - \alpha log\pi_\phi(\tilde{a}_\phi(s,\xi)|s)\right] \tag{2.40}$$

$$\max_{\phi} \mathop{\mathbb{E}}_{s\sim D,\xi\sim\mathcal{N}} \left[Q_{\theta_1}(s, \tilde{a}_\phi(s,\xi)) - \alpha log\pi_\phi(\tilde{a}_\phi(s,\xi)|s)\right] \tag{2.41}$$

Algorithm 2 Soft Actor Critic

```
1: Initialize policy parameters $\phi$, Initialize Q-value parameters $\theta_1, \theta_2$, and set target parameters equal to main parameters $\theta_{targ1} \leftarrow \theta_1, \theta_{targ2} \leftarrow \theta_2$, V-value paramters $\psi_a$, $\psi_b$, $\tau$ soft update parameter, $\Lambda_V$, $\Lambda_Q$, $\Lambda_\pi$ (objective functions for v-value, Q-value and actor networks respectively), learning rates, Empty Replay Buffer $\mathcal{D}$.
2: while Not converged do
3:     for each iteration do
4:         for each environment step do
5:             Observe current state $s_t$ and select action $a_t \sim \pi_\theta(\cdot|s)$
6:             Execute and observe next state $s'$, reward $r$, done signal $d$ indication for goal achievement.
7:             Store experience $(s, a, r, s', d)$ in $\mathbb{D}$
8:         end for
9:         for each update step do State Sample experience batch $\mathcal{B} = \{(s, a, r, s, d)\} \sim \mathbb{D}$
10:            $\psi_a \leftarrow \psi_a - \lambda_V \nabla_{\psi_a} J_V(\psi_a)$
11:            $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
12:            $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J_\pi(\phi)$
13:            $\psi_b \leftarrow \tau\psi_a + (1-\tau)\psi_b$
14:        end for
15:    end for
16: end while
```

Chapter 3

Related Work

This chapter aims to summarise ideas and algorithms that contributed to the creation of the proposed solutions in this work. The papers that will be presented express the ideas (subfields) of `Hierarchical learning`, `Curriculum Learning`, `Hindsight Relabeling ER`, `Neural Network`. Neural Networks are used both as approximators and also for generation and discrimination.

Some of the ideas can be mixed together in one solution, and sometimes the same method can act as two of those together, which makes it complicated to make a definitive distinction. Having that said, the sections are written such that a paper is under a subfield which its novelty expresses the most.

3.1 Universal Value Function Approximators (UVFA)

The idea of UVFA is presented by Schaul et al. in [7]. First, let us define the generalization of MDP formulation. `UMDP`, Universal Markov Decision Process, is a generalisation of `MDP`, where instead of having only one goal, we have a **goal space**, which means a set of possible goals. The problem formulation is the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, P, \mathcal{G} >$. The difference is that now $\mathcal{G}$ is the goal space that contains all the possible goals in the problem while $\mathcal{G} \subseteq \mathcal{S}$. Now the reward function $\mathcal{R}$ is defined as $\mathcal{S} \times \mathcal{G} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$. That is, from the accumulated experience tuple of (state, goal, action, next state) to a reward r.

The general value function $V_g(S)$, which depends on the policy π parameterised by θ, represents the utility of any state s in achieving a given goal g following the policy. Each value function is represented by a pseudo-reward function that occupies the place of the real rewards in the given problem, and represents a piece of knowledge about the environment, which is the way to evaluate a specific aspect of the environment. Finally a collection of general value functions provides a powerful form of knowledge representation, which is very useful.

The function approximator can be either a linear combination of features of a neural network (Also in our implementation we use neural networks as function approximators). The function approximator makes use of the state space structure to efficiently learn the value of the observed states, and further generalise to the value of similar unseen states. The UVFA [7] $V(s, g; \theta)$ extends the idea of **value function approximators** to both states s and goals g. In the same way $Q(s, g, a; \theta)$ extends the idea of **Q-value function approximators**.

For every goal $g \in \mathcal{G}$, a pseudo-reward function $R_g(s, a, s')$ is defined. For any policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ and each goal g, they define a general value function that represents the expected cumulative pseudo-discounted future pseudo-return:

$$V_{g,\pi}(S) := \mathbb{E}\left[\sum_{t=0}^{\infty} R_g(s_{t+1}, a_t, s_t) \prod_{k=0}^{t} \gamma_g(s_k) \middle| s_0 = s\right] \tag{3.1}$$

while actions are generated according to the policy π. The action-value function looks as following:

$$Q_{g,\pi}(s, a) := \mathbb{E}\left[R_g(s, a, s') + \gamma_g(s') \cdot V_{g,\pi}(s')\right] \tag{3.2}$$

Schaul et al. presents in [7] two possible architectures to handle working with both states and goals as inputs to the approximator. The one we are going to use is referred to as "concatenated architecture", where the state and the goal are concatenated, and together they act as input, as can be visualised in Figure 3.1.

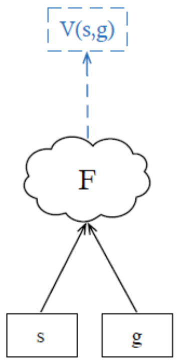


Figure 3.1: Concatenated architecture

Each goal allows an optimal policy $\pi_g^*(s) := \text{argmax}_a Q_{\pi,g}(s, a)$. The corresponding optimal value functions are then $V_g^* := V_{g,\pi_g^*}$ and $Q_g^* := Q_{g,\pi_g^*}$. Hence, the approximators will be as follow:

$$V(s, g; \theta) \approx V_g^*(s) \tag{3.3}$$

$$Q(s, a, g; \theta) \approx Q_g^*(s) \tag{3.4}$$

Those are parametrized by $\theta \in \mathrm{R}^d$ and approximate the optimal value function with respect to both state space S and goal space $\mathcal{G}$. In our implementation, the same idea will be applied to the `Actor` network in the case of SAC.

3.2 Experience Relabeling

The paper [2] suggests an important method, called `HER - Hindsight Experience Replay`, for exploiting episodes that are not successful. HER is a technique that allows sample-efficient learning from sparse binary rewards, meaning no need for complicated reward engineering. It can be applied together with an off-policy RL algorithm and it can be viewed as an implicit curriculum. This method is applicable in cases of multiple achievable `goals` as in our UVFA. In practice, when an episode has been collected, one of the states in the episode is chosen to be stored as if it was a successful episode to this new chosen goal. There are several different strategies for choosing the intermediate state among the rest. The strategy that showed the best performance is to choose a random state within the sequence. This is called the "future" strategy and we use it in our implementation.

Experience relabeling is a very useful technique and it is adopted in several more papers presented here under other subfields.

3.3 Hierarchical Learning

As mentioned, the idea behind Hierarchical Learning is to have several hierarchies that collaborate to compose the solution of the problem together. Each hierarchy has its own resolution of the problem, while the lower one interacts with the environment and its resolution is the primitive actions, the lowest possible. That allows breaking down a problem into sub mini problems that are easier to master. The algorithm as a whole then is able to learn faster.

A `Hierarchical Learning` framework with `Actor Critic` is proposed in `HAC` [16], as a solution to sample efficiency. Many DRL algorithms force agents to learn at the lowest level of temporal abstraction, performing only primitive actions. Complex sequences of actions become very difficult and slow to learn, especially with sparse rewards. It takes longer for the `Q-values` back propagation to take action and the exploration is restrained.

In the proposed algorithm, agents learn to divide tasks that involve continuous state and action space to different tasks that correspond to different time scales. It is achieved by agents that learn multiple policies in parallel. Subsequent hierarchies learn to break down tasks corresponding to increasing time resolution. Figure 3.2 exemplifies the different goals generation in the different 3 hierarchies. The distances between the vertical lines express the amount of time the agent has to accomplish each subgoal. The lowest level can be interpreted as sub goals that require one action, while the highest breaks down the ultimate goal with relatively large time resolutions. This approach has both the advantage of that the learned policies are length-limited (shorter policies are learned more quickly) and also, it allows high level exploration. Each hierarchy is an `Actor-Critic` network. Each agent learns policies that are limited time-wise.

The algorithm uses DDPG, UVFA and HER. The input of each `Actor` network is the current state and higher level goal, while the output is a particular time-scale action that can be referred to as `subgoal`. Each `Actor` network has its own `Critic` network and replay buffer to learn a near optimal policy. Each `critic` network approximates the `Q-function`

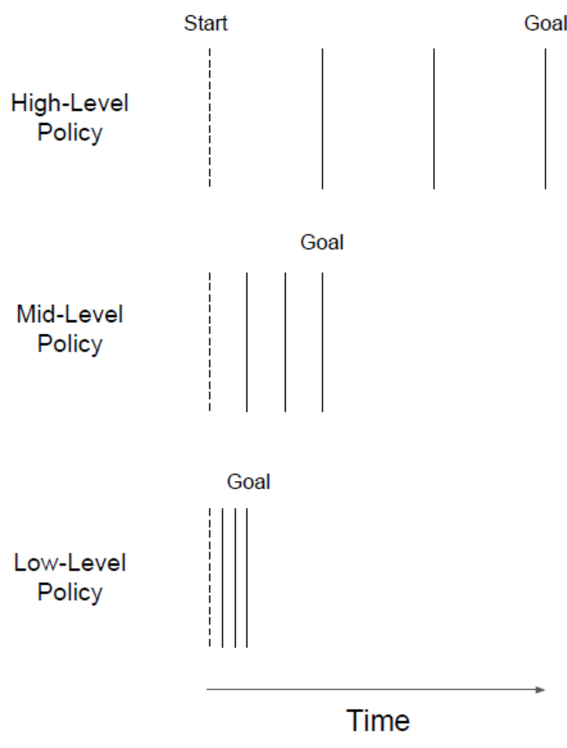


Figure 3.2: *HAC* general concept

for the associated policy using `Bellman` equation as target.

It is expected to be easier for the agents to learn multiple shorter policies in parallel rather than one long policy. The subgoals that the `Actor` network needs to learn should be efficient (achieve its high level goal in as minimum amount of actions as possible), while suggesting goals that are achievable by the lower level actor in a limited number of steps (low level horizon). Those two objectives are contradicting, hence, a coordination between the higher level and lower level is needed. To address this need for coordination `HAC` proposes two measures: First, all experience transitions passed to the replay buffers of subgoal actor networks contain actions that were actually achieved by the consequent, lower level actor network within the number of actions limit. Secondly, `HAC` penalises when subgoals that are proposed were not achieved. If layer i proposed a subgoal that was not achieved, layer i receives a negative reward with no discount factor, making it not dependent on the `Q-value` of a different state.

`HAC` shows good performance, however it also suffers from a problem of **non-stationarity** when different hierarchies learn in parallel, building on non-optimal estimations of lower levels. The paper [17] tries to tackle the non-stationarity problem. It suggests that `RL` can be used to learn all policies in parallel if each level above ground level has a way to simulate a transition function that uses the optimal versions of lower level policies. The suggested framework allows the user to simulate a transition function that uses an optimal lower level policy hierarchy. The algorithm is implemented basing on two concepts that express HER:

1. *Hindsight Action transitions*: As explained previously, when a layer $i-1$ is unsuccessful

over several attempts in achieving a subgoal g_i proposed by layer i, but it achieves instead another goal g'_i, than, layer i receives a transition that includes g'_i as the action. That is in practice, assuming that the lower level $i-1$ policy is optimal.

2. *Hindsight Goal transitions*: One of the states reached in hindsight is used as the goal state in the transition instead of the original goal.

The most significant drawback of that approach is that level i can only learn Q-values for subgoal proposals (its actions in practice) that are relatively close to its current state and will ignore the Q-values for all subgoal actions that require more than $\mathcal{H}$ (horizon) actions. Another problematic phenomenon is that subgoal hierarchy ignores the lower level abilities, and so the output subgoals created might be suboptimal. To overcome this disadvantage, an approach for **subgoal testing transitions** is proposed. That is to help a subgoal layer to learn whether the subgoal it proposed is reachable within the horizon H steps of the lower level with its current policy. If it not reachable, the transition will contain a penalty of $-\mathcal{H}$.

Another algorithm called `HIRO` suggested in [18] to tackle the problem of non-stationarity, by **off-policy correction**. The approach is to relabel g_t for past high level policy experiences in order to make the observed action sequence more likely with the current lower level policy. Having $a_{t:t+c-1} \sim \pi^{lo}(s_{t:t+c-1}, g_{t:t+c-1})$, the goal that should be stored is $\tilde{g}_{t:t+c-1}$, the one that maximizes $\pi_{lo}(a_{t:t+c-1}|s_{t:t+c-1}, \tilde{g}_{t:t+c-1})$ with the current low policy π^{lo} out of 10 candidates. That is approximated by 3.5. Eight candidates are sampled from a `Gaussian` around $s_{t+c} - s_t$, while one candidate is always $s_{t+c} - s_t$, which is equal to g_t if the algorithm performs correctly, and trivially, g_t itself is the tenth and last candidate.

$$log\pi_{lo}(s_{t:t+c-1}, \tilde{g}_{t:t+c-1}) \propto -\frac{1}{2}\sum_{i=t}^{t+c-1} ||a_i - \pi^{lo}(s_i, \tilde{g}_i)||_2^2 + constant \tag{3.5}$$

The paper [19] presented by Wang et al. also tries to tackle the problem of non-stationarity of high-level training for decision making - `AGILE (Adversarially Guided Subgoal Hierarchical Generation)`. The proposed approach is to adversarially enforce the high-level policy to generate subgoals that are in accordance with the momentary instantiation of the low-level policy. That aims to enhance high-level policy's knowledge of the low-level's ability. The generator is the high level policy that learns to generate subgoals following a compatible distribution with the current low-level policy. A discriminator network is used to distinguish a generated subgoal that may not be reachable by the low level policy from a relabeled subgoal that is known to be reachable.

The framework proposed has 2 hierarchies. The high-level modulates the behaviour of the low-level policy by intrinsic rewards for reaching the generated subgoals. The high-level policy aims to maximise the extrinsic reward r^h_{kt} that is defined in 3.6. The low-level policy aims to maximise the intrinsic reward provided by the high-level policy, that is measure by subgoal reaching performance: $r^l_t = -||s_t + g_t - s_{t+1}||_2$.

The architecture used is of `TD3` for each of the two hierarchies. One objective of the subgoal generator is to maximize the expected return induced by a deterministic policy in 3.7.

$$r_t^h = \sum_{i=t}^{t+k-1} r_i^{env}, t = 0, 1, 2, ... \tag{3.6}$$

$$J_{dpg} = \mathbf{E}_{\mathbf{s}\sim\mathcal{D}}[Q_h(s, g)|g = G(s; \theta_g)] \tag{3.7}$$

The subgoal generation network $g = G(s; \theta_g)$ maps from state space to subgoal space (not sampled from random noise as in vanilla `GAN`). $\mathcal{D}$ is the replay buffer with the high level action relabeled: g_t of the high-level transition $(s_t, g_t, \Sigma_{i=t}^{t+k-1} r_i^{env}, s_{t+k})$ is relabeled with $\tilde{g}_t$ to maximise the probability of the created low-level sequence of actions: $\pi_l(a_{t:t+k-1}|s_{t:t+k-1}, \tilde{g}_{t:t+k-1})$. This probability is approximated by maximising the log probability in 3.8.

$$log\pi_l(a_{t:t+k-1}|s_{t:t+k-1}, \tilde{g}_{t:t+k-1}) \propto -\frac{1}{2} \sum_{i=t}^{t+k-1} ||a_i - \pi_l(s_i, \tilde{g}_i)||_2^2 \tag{3.8}$$

A major difficulty in HL is that higher levels should suggest **effective subgoals**, without actually knowing if they are reachable or not, since this information can be collected only by the lower level after having sufficient exploration and training. The papers [20], [21] by Zhang et al. present the algorithm `HRAC`, which tries to tackle this problem. The term "Shortest transition distance" refers to the minimum number of steps needed to reach a target state from a start state. It can be achieved by minimising the expected first hit time over all possible policies. The high-level action space can be restricted from the whole goal space to a k-step adjacent region centred at the current state.

The algorithm requires an `Adjacency network` training from `Adjacency table`: explicitly memorise the adjacency information by constructing a binary k-step adjacency matrix of the explored states. The adjacency matrix has the same size as the number of explored states, and each element represents whether two states are k-step adjacent. The adjacency network learns a mapping from the goal space to an adjacency space, where the Euclidean distance between the state and the goal is consistent with their shortest transition distance.

Another interesting paper by Kim et al. presents the algorithm called `HIGL` [22], which is similar to `HRAC` but it evaluates both the reachability of states, and their **novelty**. The novelty score is higher for novel states dissimilar to the ones the predictor network has been trained on. The adjacency Network discriminates whether two states are k-steps adjacent or not. It learns a mapping from goal space to an adjacency space, by minimising the contrastive-like loss. Then it is possible to approximate the shortest transition distance. The algorithm does **Landmark sampling**: coverage based sampling and novelty based sampling. Coverage based sampling means to sample farthest points from a wide range of visited states from the `Experience Replay`. The distance is measured in the goal space as norm2. As for novelty-based sampling, they introduce a priority queue of a fixed size ordered by novelty. The novelty of a state decreases constantly, meaning it should be updated all along. previously sampled similar states are discarded.

After **Landmark sampling** comes **Landmark selection** to detect the urgency of the landmark since not all the landmarks are helpful. First, build a graph of landmarks, then run the shortest path planning to a goal in a graph. Nodes in the graph consist of current state, a goal, and landmarks. Each edge is weighted with the distance that is estimated by low-level goal conditioned value function. When the calculated distance is too big, the edge is being discarded, since distance estimation via value function is only locally accurate. Value iteration is used as shortest path planning, and the first landmark is selected to be the goal.

3.4 Curriculum Learning

`Curriculum Learning` (CL) aims to address the sampling inefficiency for off-policy methods problem to accelerate RL training procedure and for some cases to improve the results, by performing the experience acquisition in heuristic systematic manner than the random. The inspiration of CL is the fact that animals and humans learn better when the experience they see is ordered in a meaningful manner, that gradually encompasses more concepts, with increased levels of complexity.

Bengio et al. claims in [23] that curriculum strategy can act like a continuation method, which helps to find a better local minima, and points out that curriculum strategy operates like a regularizer, as reported in experiments. For convex criteria curriculum strategy can speed-up the convergence to the global minimum. In an older paper 1993, Elman presents the notion of *starting small* [24] and makes the statement that this strategy enabled humans to learn what might otherwise prove to be unlearnable.

One difference between HL and CL is that HL is built such that both agents learn different resolution of solution to the problem, and eventually in test time, the higher policy is part of the solution. On the other hand, CL might involve two agents so the one aims only to accelerate the learning of the actual agent, and the first does not form part of the ultimate solution in test time. Also sometimes the implementation of curriculum does not imply another agent, but the goals that are suggested, are characterised by some heuristics so the learning is accelerated. The suggested algorithms in this work combine the two ideas, and let both the agents learn in a curricular way.

While in first years the focus of CL was mainly on constructing manually a sequence of tasks of **increasing difficulty**, in more recent papers the focus is on the **automatic curriculum generation**. A practical (automatic) curriculum learning method addresses training examples ordering and sampling modification based on the order. A typical guideline of curriculum algorithm would be:

1. Curriculum design.
2. Evaluation metric design - how easy or difficult the current target is?
3. Sequentially Increasing difficulty level tasks (training).

A classic paper in the field of CL is [25] *Teacher and Student Curriculum Learning (TSCL)*, where a teacher agent chooses automatically which task to give to the student which allows the student agent to learn gradually more complex tasks. At each time step the teacher selects a task from a list of possible tasks, the student trains on the task and returns a score. The goal of the teacher for the student to succeed in achieving the ultimate task goal.

A big challenge is for the teacher to estimate the level of the student, to pick tasks in the right level of difficulty and balance the exploration exploitation trade-off. The problem is that the only learning signal the teacher has is from the noisy scores of the student, which makes it harder to track the learning curve of the student. To overcome this, three approaches are presented and tested: Firstly, the naive approach, where an average over K times the students tries to solve the problem, then use the regression coefficient of the learning curve as a reward in a non-stationary bandit algorithm. Secondly, FIFO buffer over last k scores, then again linear regression to estimate the slope of the learning curve for each task, with respect to time steps. While those two options require exploration hyper parameters tuning, the third option takes inspiration from `Thompson sampling`, where k last rewards for each task is kept in a buffer, then to choose the next task, a recent reward is sampled for each task, and whichever yields the highest reward gets selected.

Another paper [26] aims to investigate the effect of the task distribution as a hyperparameters in the `Gradient-based Meta-Learning` framework. When the distribution contains a large variety of tasks, most likely an underfitting problem will occur, since the agents are incapable to specialise in neither of the tasks reaching a good performance. On the other hand, low variety might lead to poor generalisation. The random sampling of tasks can also harm the ability to learn, in cases where qualitatively different tasks are chosen. The idea is to address the problem of meta-overfitting by explicit optimization for task distribution.

The proposed method is called `Meta-ADR (Active Domain Randomization)` and it was proposed by Mehta et al.. The algorithm, as can be visualised in figure 3.3, helps the agent to learn a curriculum of tasks, instead of uniformly sampling the given tasks set. A discriminator that was trained over pre-adaptation and post-adaptation sequences, is used to learn task difficulty, via rewards, which are used for the particles training. Intuitively `ADR` is optimised to find environments where the same policy results with different behaviour.

Another interesting paper [27] by Held et al. presents `GoalGAN - Automatic Goal Generation for Reinforcement Learning Agents`, an adversarial training approach for subgoals generation, which are always in the appropriate difficulty of the agent, producing a curriculum. The method includes a dynamic modification of the probability distribution from which goals are sampled, which assures that the difficulty level of the generated goals is appropriate.

The algorithm formulation directly motivates the agent to train on tasks that challenge its capabilities, which reflects the concept of **Intrinsic Motivation**. In addition, the algorithm is based on **Skill-learning**, which is an approach where an agent can reuse skill, an improvement over learning from scratch. The third idea that the algorithm is based on is `Curriculum Learning`.

The algorithm uses an indicator function, with a tolerance and a distance metric, such

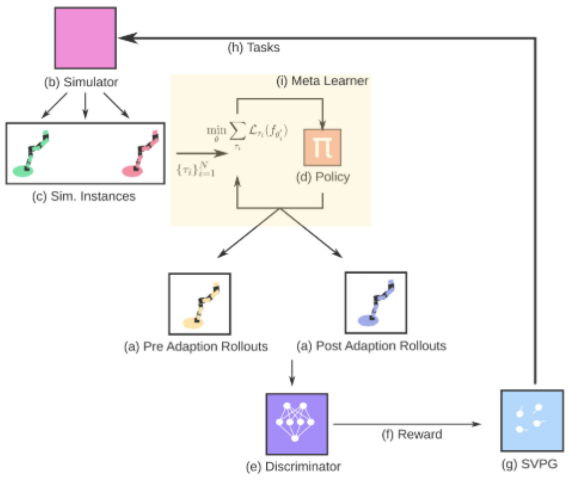


Figure 3.3: *Meta-ADR*

that when reaching a certain distance from the target within the `T` time steps limitation (horizon), it counts as 1, otherwise 0. It applies to problems with binary and sparse reward, which resembles real world problems. The goal space is continuous. They define an objective of equation 3.9, named Coverage function, which should to be optimised.

The first step of the algorithm is goal labelling as **Goals of Intermediate Difficulty** ($GOID_i$) or not. $GOID_i$ are goals that during iteration i, yield a long term return within a certain range of R_{min} and R_{max}, as described in 3.10. Goal sampling during training is then uniform over the set of $GOID_i$. In this way, the agent is trained with goals that will result in some reward, but has not yet been mastered (not receiving the maximum reward it can get). R_{min} and R_{max} can be seen as the minimum and maximum probability of reaching a goal in `T` time steps. The algorithm suggests to first estimate the label $y_g \in \{0, 1\}$ that indicates whether $g \in GOID_i$ for all goals g used in the previous training iteration, then use these label to train a generative model from where goals sampling for the next iteration is possible. The label estimation is done by calculating the fraction of success out of all trajectories that contain g in the previous iteration. Then the result can be passed through 3.10 threshold to be labelled as 1 or 0.

$$\pi^*(a_t|s_t, g) = \arg\min_{\pi} \mathbf{E}_{g\sim P_g(\cdot)} R^g(\pi) \tag{3.9}$$

$$GOID_i := \{g : R_{min} \leq R^g(\pi_i) \leq R_{max}\} \subseteq g \tag{3.10}$$

The second component of the algorithm is Adversarial Goal generation, using `Goal GAN`. `Goal GAN` is a Generative Adversarial Networks with a modification that allows training with both positive examples and negative examples (factuals and counterfactuals). Negative examples mean they were sampled from a distribution that does not share support with the desired one. GANs also allow scaling to high dimensional goal spaces (such as images). The `Goal Generator` $G(z)$ is trained to uniformly output goals from noise vector z, that are in $GOID_i$, using `Goal Discriminator` $D(g)$. The optimization is done similar to `Least-Squares GAN (LSGAN)`, with the modification for negative examples.

Let us emphasize the a main difference between `AGILE` (the hierarchical adversarial algorithm that was presented under 3.3), with the recently presented `Goal-GAN`: `Goal-GAN` does not have a condition on the observation, instead, its generator stands alone. The GAN and the policy are separated and sequentially trained. `AGILE` generator on the other hand, is a substitute of the actor network, and policy update is performed directly through the adversarial loss and policy loss concurrently.

3.4.1 Reverse Curriculum Generation for Reinforcement Learning

One way of automatic curriculum for sparse environments is `Reverse Curriculum` [28]. The algorithm suggests a framework where a robot is trained in "reverse", from the goal backwards. In practice it means to reach the goal from a set of starting states that are increasingly far from the goal. Those "promising" starting states are located around the target (or states that are similar to the goal state), which means it is easier and more probable to reach the target from them. When the agents master the task of reaching the goal from "good starts", it is now possible to spread more, and learn how to reach from states that are close to the "good starts", to the promising state. That can be seen in a way where the "promising state" becomes the new goal, for other "promising states". In this way, gradually the agents learn to reach different goals (or states). This way of training assures that the given goals are reachable with intermediate difficulty. Intermediate difficulty for most cases is defined to be tasks that the agent has not mastered yet, but has at least some success.

Reverse Curriculum Learning is expected to accelerate the learning since the agents should reach the target relatively fast, which grants him with high reward. That is a strong learning signal which is beneficial for the learning.

This paper suggests a discrete-time finite-horizon MDP by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0, \mathcal{T})$. Most of those were already explained in section 2. $\rho_0 : \mathcal{S} \to \mathbb{R}_+$ is a start state distribution, r and T are the reward function and the horizon respectively. The starting state distribution ρ_i changes in every learning iteration i, while evaluation is done on the uniform distribution of the states, as in the original problem. The general problem that is considered then is reaching goal space $\mathrm{S^g} \subset \mathrm{S}$ from any start state $\mathrm{S^0} \subset \mathrm{S}$. Having the reward a binary one in case reaching a state the belongs in $\mathrm{S^g}$, implies that the return associated with every start state s_0 is the probability of reaching the goal in the T horizon limit.

The suggested algorithm relies on 3 assumptions:

1. *It is possible to resent the agent into any start state $s_0 \in \mathrm{S}$ at the beginning of all trajectories.*
2. *At least one state s^g is provided such that $s^g \in \mathrm{S^g}$.*
3. *The Markov Chain induced by taking uniformly sampled random actions has a communicating class including all start states $\mathrm{S^0}$ and the given goal state s^g.*

To find those ”good starts”, a procedure called `SampleNearby` is performed concurrently during the learning. The agent performs T (horizon) random actions from the goal state, which bring it to new states, in the vicinity of the goal state. Several different series of actions are done, so those states which the probability of reaching the goal from them is high, are collected to the list of good starts. Then the distribution of those good starts is updated with the newly found states. All those states visited during the rollouts are guaranteed to be feasible and can then be used as start states.

The ”good start” list in the beginning contains only the goal state s^g. The curriculum is expressed by the fact that the suggested starting states are uniformly sampled from that list. The algorithm selects only states where the expected return from them is between R_{min} and R_{max}, which assures those are not yet mastered but also feasible - not too hard and not too easy [29]. In theory, to calculate the expected return from them, it is needed to perform some trajectories from each of them, however to avoid this inefficiency, they use the trajectories collected by the policy training trajectories.

Gradually, this region of ”good starts” grows, and hopefully covers all possible starting states in the problem. They showed that `Brownian Motion` from that ”good starts” list, performed way better than the option of only uniformly sampling from the state space.

Chapter 4

Algorithms

4.1 Motivation

Reinforcement learning algorithms still have a lot of drawbacks that prevent them being widely used, as explained in chapter 1. The idea in this master thesis was to create an effective and efficient algorithm that will be able to solve complex environment tasks.

An important work that has been done last year (2021) by the colleague Rafel Palliser Sans, presented the algorithm `Learning Recursive Goal Proposal`, **LRGP**. This algorithm was also meant to solve RL complex environment problems with **Hierarchical Learning** using two policy levels - high and low. The high policy aims to suggest meaningful subgoals in the way from a given state to a goal, so the low policy only has to learn short sub-trajectories between the suggested subgoals. The low level is the one that interacts with the environment exclusively, performing (primitive) actions, which result in the agent's state changes. The actions of the high hierarchy are in practice the goals of the low hierarchy, and they are added to the `goal_stack` in every proposal in First In First Out (FIFO) technique. The high agent keeps suggesting subgoals **recursively** as long as the current subgoal is not reachable by the low agent, and the limit of allowed subgoals does not exceed the high agent's horizon. The reachability of a goal/subgoal is verified with `is_reachable` set that stores the possible movements of the agent within the limit of the low horizon from past experience. The task of the algorithm is to empty the `goal_stack`, meaning the episode's ultimate goal was achieved (it is the last one to be erased from the stack). For sample efficiency and learning acceleration algorithm uses HER.

LRGP showed relatively high performance and was able to outperform some State of the Art algorithms in the presented environments. However it had some points for improvements.

One of the challenges is the incomplete space of the environments: some states are not feasible in the sense that the agent is not able to be in those. For more details regarding the environment, please refer to section 5.1. The high level of `LRGP` suffers from impossible subgoal suggestions, since it does not learn the environment directly. The incomplete space is handled by storing a set of all visited states as **allowed** subgoals for proposals. When the high policy suggests a subgoal that does not appear in this list, it will be eliminated and another attempt of the high policy to suggest an appropriate subgoal is done, while adding some noise. In practice, when a suggested subgoal is not allowed it leads to recursively suggesting locations that are close to the original proposal until an acceptable one is proposed, where frequently several suggestions are needed until an allowed state is proposed as a subgoal.

Another issue is that the success rate by the end of the learning process keeps oscillating around 0.89 success rate, meaning there still exist a gap to improve. That is further worsen when expanding the goal state dimensions to 3 dimensions, to be identical to the state space. In that case the performance of **LRGP** seems to degrade.

For these reasons we aspired to offer a **Hierarchical** algorithm, that integrates several common component with `LRGP` (such as `goal_stack` and `is_reachable` set), while basing on other different concepts, that will allow higher success rate, while also avoiding the **unfeasible subgoals proposals**. To tackle those objectives we offer algorithms that adjust the idea of **Reverse Curriculum Learning** to Hierarchical learning.

In this chapter we will present the integration of the Reverse Curriculum Learning idea in the hierarchical learning framework in our context. Then we will present the two main developed algorithms in this thesis: `Reverse Curriculum Recursive Learning (RCRL)` and `Reverse Curriculum Vicinity Learning (RCVL)`.

4.2 Reverse Curriculum Learning: Adjustments to the Hierarchical framework

Under section 3.4.1 we went through the idea suggested by Carlos Florensa et al. in [28] in details. In the next subsections two algorithms that combine this idea of `Reverse Curriculum Learning` with the idea of `Hierarchical Learning` are presented. Florensa in his work [28] proposes to maintain a list of "good starts" which are state with high potential for success. Those are collected by sampling states nearby the goal state. If those states are reachable from the goal state, meaning that the goal is reachable from them in reversible environment.

The subgoals are uniformly sampled from this set of "good starts". The curriculum is expressed by the difficulty to reach the goal from them, which is estimated by the return from them over number of experiences. The average in the binary reward environment is equal to the probability of success. Those with 0 return are too hard, while those with 1 mean that they are mastered already, too easy. This can be visualised as if it was the area around the goal that is being learned by the agent, gradually it expands, until hopefully the whole state space has been visited by the agent in mastering level.

Those "good starts" are in practice milestones along the way from the state to the goal, that divide the episode path into several **mini-trajectories** that are expected to be shorter than the original path required. The hierarchical architecture facilitates the learning of the low agent, by making it only the mini-trajectories. Shorter sequences to learn are easier (statistically more probable) to learn.

The learning of the different levels is done concurrently, which has its implications: the high level does not have interactions with the environment and the information it gets is limited and based on the lower level. That is - the high level learning is strictly subjected to the low level learning. In addition, when the high level proposes unfeasible subgoals, or overly hard ones, the lower level is more prone to fail.

For all of the algorithms, the terms of "close" and "far" are quantified and measured by the number of steps needed to move from one state to the other. We do not use the euclidean distance, since it would not necessarily reflects the reality in the environments (think of two close states that only one state is between them, but it is unfeasible).

4.3 Main common components for both suggested algorithms

4.3.1 Low policy

The low hierarchy can be described as a UMDP with state space of $\mathcal{S}$, action space $\mathcal{A}$, goal space $\mathcal{G}$, which is equal to the $\mathcal{S}$. Note that goals of the low agent are unstacked from `goal_stack` and are drawn by the high policy, rather than the environment. That implies the goal space perceived is smaller from the whole space, facilitating the low policy learning. The reward short term reward is defined to be 0 for the step that achieves a mini trajectory goal and -1 for any step (binary sparse reward). Note that the mini trajectory goal is different from the ultimate episode goal. The discount factor is defined to be $\gamma = 1$ having the long-term reward upper bounded by zero. The negative reward of -1 for each step encourages the agent to learn how to go through a trajectory with the minimum steps possible.

The algorithms make use of `is_reachable` set, which is a set of possible transitions within `low horizon` steps in the form of (state, goal). This set grows larger and larger, especially in the beginning of the learning, while a lot of exploration is done.

The low policy is implemented by DDQN with HER, using `Future strategy` (see section 3.2) for goal relabeling. This selection of DDQN is following the study that was performed in the `LRGP` thesis, which showed it fits to the environments we test in the thesis.

4.3.2 `is_reachable` set

The low agent also stores in its property the set of $(state, goal)$ tuples, which are collected as possible during one `low horizon`. When the low agent performs its steps, the visited states are stored in the a list. Then after a `low horizon` actions of the low agent, were made, this list is rolled out and each combination of two states s_1, s_2 where s_1 occurred before s_2 is stored in the `is_reachable` set. This set is being used to verify if the current proposed goal is reachable within one `low horizon` by the lower policy. Based on this information, a new subgoal/starting state is proposed: if the tuple of $(current\_state, current\_goal)$ exists in the set, then the low level gets into action. Otherwise, another subgoal will be proposed by the high level.

4.3.3 High policy

The high policy aims to suggest promising starting states (subgoals), and it is implemented differently in the different algorithms proposed. The problem can be defined as UMDP with state space $\mathcal{S}$, goal space $\mathcal{G}$ which is the same as $\mathcal{S}$. Action space $\mathcal{A}$ of the high policy is the same as $\mathcal{S}$ since its part is to propose starting states. The transition probability P depends on the lower policy, since the lower policy is the only to actually interact with the environment.

The states that are perceived by the high level are the first and last in every mini-trajectory, meaning states that `low horizon` consecutive steps are between them. Those are high level steps. That is why a high level is able to "capture" the big picture. The high level steps are stored from the beginning of the episode until the end of it, and then they are rolled out to be store in the high hierarchy ER.

The high level agent is implemented as variation of SAC: `RCRL` uses the explicit policy representation while `RCVL` uses the value function approximator for most cases, and the explicit representation of the policy in cases where we have a goal that we don't know its vicinity yet. SAC for the high policy was chosen in a similar way to the lower level following `LRGP` thesis study, which showed it fits the environments we test in the thesis.

The **short-term reward** equals to -1 for each goal proposal, resulting with a long time reward of minus the quantity of subgoals suggested from state to goals. That makes the long-term reward be the "cost" (negative) of proposed starting states number in an episode, or number of subgoals to be suggested. Therefore learns to minimise the number of subgoals suggested, leading to minimising also the number of steps that should be done by the low hierarchy agent.

Long-term return estimation

During an episode, we store the "high level steps" done, and after the episode is finished we roll out this list and create several tuples to store in the ER. In between the high level steps we have the low level steps (maximum of `low horizon` steps), which are a black box to the high level. The idea of `Hindsight Action Transition`, that is presented in [17] and was used in [30], means that the high policy learns under the assumption that the low policy is optimal. That helps to stabilise the learning though having the low policy changing.

When rolling out the "high level steps", we relabel the actions to be in a way that the created sequence has successful sub-trajectories. Sub-trajectory is a partial path (s, a, g) of the existing sequence, where always s is visited not after a and a is visited not after g (either before or they are the same). Then there are two optional approaches. First one is the `Monte Carlo` empirical approach, and the other is based on the long term reward estimation. The second one was used only for a variation of the algorithm `RCVL`, that is presented in 4.5.4.

- **Monte Carlo** based approach, where the number of subgoals proposed between state s and goal g is the actual number of high level steps between them. Let us look at an example: (s_1, s_2, s_3, s_4) visited high level steps during an episode. We will store all the following tuples in ER of high policy:

  1. $(s_1, s_2, -1, s_2)$ - one subgoal was required to be proposed to reach from s_1 to s_2 via s_2.
  2. $(s_1, s_2, -2, s_3)$ - two subgoals were required to be proposed to reach from s_1 to s_3 via s_2.
  3. $(s_1, s_2, -3, s_4)$ - three subgoals were required to be proposed to reach from s_1 to s_4 via s_2.
  4. $(s_1, s_3, -3, s_4)$ - three subgoals were required to be proposed to reach from s_1 to s_4 via s_3.
  5. $(s_2, s_3, -1, s_3)$ - one subgoal was required to be proposed to reach from s_2 to s_3 via s_3.
  6. $(s_2, s_3, -2, s_4)$ - two subgoals were required to be proposed to reach from s_2 to s_4 via s_3.
  7. $(s_3, s_4, -1, s_4)$ - one subgoal was required to be proposed to reach from s_3 to s_4 via s_4.

  We can see that this exploits a lot of the collected experiences, which reflects the sample efficiency. We accumulate much faster experience that otherwise would have required more actual transitions, meaning more episodes. The value networks can then learn to predict the reward (cost) of minus the quantity of subgoals proposed. A proposed subgoal requires in practice low horizon steps of the low level.

- **Value estimation** based approach. In our UVFA implementation of SAC, we have a $V(s, g)$, which is the approximation of the long-term return going from state s to goal g following the policy. In addition we have two `Q-value` networks (not including their target networks) $Q(s, a, g)$. When rolling out the high level experience to store in the ER at the end of an episode, we assign the reward for (s, a, g) of the high level. Here we aspired to store the **value estimation** using the value approximators. For every movement from state s to goal g via action a, we compare $Q(s, a, g)$ (state, subgoal, goal) to $V(s, g)$ (state, goal). If the first is higher (higher value, lower cost), that means that in order to move from state s to state g, it is better to pass the way via a, than the currently policy. Then we store in the ER the tuple (s, a, v_{approx}, g). We expect the algorithm to converge to the optimal path, similar to the previous approach that does empirical experience storing. It is important to say that also in this variation of experience storing, we perform the presented calculations for each of the (s, a, g) tuples as in the previous approach. In the sense of sample efficiency it is the same, however, the real problem with this approach is that the learning is very difficult. Further explanation is brought under section 4.5.4.

For both main algorithms suggestions `RCRL` and `RCVL` the used approach is the first one (monte carlo). `RCVL` has a variation that uses the second approach.

In the case where the high policy suggests a subgoal that is identical to current goal or current state, than we store to the high policy ER a "penalty" of negative `high horizon` for this suggestion, meaning it is a worthless suggestion, as if it was leading to an unsuccessful episode.

4.4 Reverse Curriculum Recursive Learning (RCRL)

4.4.1 The idea

Combining the idea of the reverse curriculum with hierarchical learning, we use the high policy to be the one to suggest those *good starts*. The idea is that having a starting state and a goal state, $[s, g]$, we want to suggest a state that is close to g, which is hopefully a *good start*, but also reduces the cost between s and g. This state should be an intermediate state between them, in the vicinity of g. The high policy is expected to learn states that g is reachable from them easily. The more the training progresses, larger areas around goals should be learned by the low policy but also, by the high policy in a more macro-resolution.

4.4.2 Algorithm flow

The proposed algorithm is presented in Algorithm 39. First we initialise the networks for the high policy and for the low policy. We iterate a number of episodes, while at each iteration we reset the environment to get a new state and goal aiming to perform an optimal path between them. In each episode we create `starting_state_list` from the goal and the state such that the first two elements in this list are the current goal state (cgs) and current start state (css) respectively (lines 1-6).

We then go into the loop (line 6) where we try to solve the trajectory between css to cgs: We check if cgs is reachable from css with **low horizon** steps and put the Boolean answer in `Reachable`. If we are exploring (probability of ϵ) we will assign True to `Reachable` without dependency of its real reachability (line 7).

In the case where the cgs is not reachable from css then the high level policy should suggest a new starting state (line 12), that will be inserted into `starting_states_list`, in the second place (line 13). That means we divided the planned trajectory into two sub-trajectories with another starting state in the middle, that is expected to be close to cgs and is closer to css than cgs. An important step here is to roll out the experience accumulated possibly over several low level runs. We use HER as explained previously. We have to roll it out and reinitialize the steps stored since when we add a new subgoal then we cut the continuity in the accumulation, and the next visits steps are not consequent to the last ones. A new accumulation of steps for both high and low level starts. We actualize cgs to be the

Algorithm 3 Reverse Curriculum Hierarchical Recursive Learning

```
Require: high agent, low agent, is_reachable(s,g) function, low horizon $H_l$, num_episodes
 1: for episode = 1:num_episodes do
 2:     s ← initial_state
 3:     g ← environment_goal
 4:     starting_states_list ← [g, s]
 5:     css ← starting_states_list[1] and cgs ← starting_states_list[0]
 6:     while True do
 7:         reachable ← is_reachable(css,cgs) or exploration
 8:         if not reachable then
 9:             if count_suggestions >max_suggestion then
10:                 Break While loop.
11:             end if
12:             new_ss ← high.propose_ss(css, cgs)
13:             starting_states_list.insert(new_ss, index=1) and roll out experiences (HER).
14:             css ← starting_states_list[1] and cgs ← starting_states_list[0]
15:         else
16:             high_state = css
17:             for low step = 1:$H_l$ do
18:                 a ← low.select_action(css, cgs)
19:                 Apply action a and observe next state css'
20:                 achieved = (css' == cgs)
21:                 store in low.experience_replay the tuple (s, a, achieved-1, s')
22:                 css ← css'
23:                 if achieved then
24:                     starting_state_list.pop()
25:                     css ← starting_states_list[1] and cgs ← starting_states_list[0]
26:                     break for loop
27:                 end if
28:                 store in low ER (css, a, bool(achieved) -1 , css', cgs)
29:             end for
30:             high_state_next = css'
31:             store in high ER (high_state, high_state_next)
32:             if length of starting_state_list == 1 then
33:                 Episode is completed. Break While loop
34:             end if
35:         end if
36:     end while
37:     Roll out experiences (HER in both policies)
38:     Every n episodes: update networks
39: end for
```

first element in `starting_states_list`, and *css* to be the second, that is the new suggested starting point (lines 14). Here we make the use of the assumption that it is possible to move the agent to a required state: we manually position the agent in the suggested starting state (css). We then iterate again the while loop.

If `reachable` is true then the low policy performs H_l (horizon of the low level) steps (lines 17-29). In the case that during those steps *cgs* was achieved, we have solved the current sub-trajectory, meaning we can erase *cgs* from `starting_states_list`, update *cgs* and *css* to now solve the next sub-trajectory and we break the inner loop of the low policy (lines 23-27). We store the high level steps in the high level ER (line 30). We start the while loop again: the next sub-trajectory that needs to be solved is from the current state with the same goal as before. However, after performing the last low level horizon steps assuming a good policy, we should be closer to *cgs*.

Recursivity stop conditions: The recursivity in this algorithm is in the starting states suggested by the `high policy`, or in other words, the division of the original trajectory s to g into several sub-trajectories. When `starting_states_list` contains only one element, then we have completed all the sub-trajectories we had, and the episode is completed (lines 32-34). Another stop condition to the recursivity is in the number of suggested subgoals (lines 9-11).

After each episode the experience accumulated for both hierarchies is rolled out with HER and all the networks are updated.

starting_state_list episode example

To simulate one example of an episode let us visualise the `starting_state_list` in figure 4.1. We initialise the environment, the agent is in state s_1 and g is the episode goal. g is not reachable so the high policy suggests state s_2 between s_1 and g (line2 in the example). The current state (*css*) is now s_2 and the goal is still g (notice that in this example the updated states are the last two elements in the sequence but in the python list those are the first ones).

We then see that g is also not reachable from s_2 then the high policy suggests another starting state s_3. Now the trajectory should be from s_3 to g. g is reachable from $s3$, so the low level performs a horizon of primitive goals towards g, hopefully arriving at g. In this example we arrive at g, which results with the deletion of it from the list, however if it was not reached, another loop trying to solve the next trajectory which is from the state that the agent has reached after the horizon primitive steps to g.

After popping out g from the list, now we update the current state (*css*) to be again s_2 from where we will try to reach s_3 (line 5 in the example). After checking, s_3 is not reachable, so the high policy suggests s_4 in between. We continue this routine until the last trajectory is solved or until one of the recursion conditions is fulfilled.

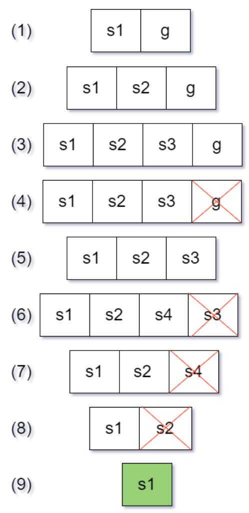


Figure 4.1: Reverse Curriculum Recursive Learning `starting_states_list` along an example episode.

4.5 Reverse Curriculum Vicinity Learning (RCVL)

4.5.1 The idea

The general idea is that the `high policy` suggests possible subgoals, from the episode goal and backwards recursively until the suggestion is reachable to the agent. An example is shown in section 4.5.4. Here, similarly to `LRGP` we use the `goal_stack` with First In First Out (FIFO) technique. The ultimate episode goal is the last one to be popped out and that is when a successful episode is finished.

Trying to tackle the problem of unfeasible subgoals suggestion in incomplete environments, we want to suggest subgoals from a **visited set of states that are located in the vicinity** of the current target. We will refer to this list of sets as `goal_list`. We collect those states during the concurrent training of the higher policy and lower policy, and possibly in another separated step of vicinity acquisition that is performed before the concurrent learning as a preliminary step. In this way we get to know the state (goal) space such that it is possible to compose trajectories from each state to any other state passing through common states (intersections) between two or more neighbourhoods as expressed in figure 4.2. The more we sample and explore and state space, the more options are available for subgoal suggestion, which implies that it is possible to get closer to the optimal solution.

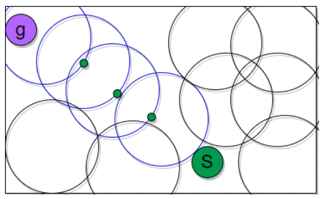


Figure 4.2: Learned neighbourhoods in the state space. Every circle is a neighbourhood around a goal. It is possible to construct a path from state s to goal g passing the common neighbours on the way. The neighbourhoods that participate are the blue circles, and the green dots are the states that form the whole path.

4.5.2 Algorithm flow

An overview of the algorithm is presented in algorithm 4. The first step is vicinity preliminary acquisition (line 1), however it is not a mandatory phase step. Then we iterate number of episodes we define, such that in every episode the algorithm first tries to create the path for the lower level, by high level sugoal suggestion, and then when the high level path is complete, and the last subgoal proposal is reachable from the current state, then the low level gets into action and goes though the constructed path by the high level.

In each episode we start with assigning 0 to the number of subgoals proposed (line 3). We get state and goal for the episode in lines 4-5. We start accumulating the high level steps in `solution` (line 6) and we stack the goal into `goal_stack` (line7). Then we get into the while loop that allows recursively subgoals suggestion and finally low level steps that will go through the mini-trajectories. If the current goal is reachable from the current state within low horizon steps, then `reachable` gets True value. In addition we explore with a probability of decaying ϵ. If it is an exploration sub-trajectory then `reachable` is assigned True (line 10).

In the case that the current goal is not reachable then it is needed to verify the number of suggested subgoals. In the case where more than `high horizon (`H_h`)` subgoals were proposed we break the while loop and the episode is terminated without success (lines 12-13). Otherwise, a new subgoal is suggested by the high policy and is stacked into `goal_stack` in case it is legal and the subgoal count grows by one (legal is when the subgoal location is different than current state and current goal) (lines 15-17).

In the other case, where `reachable` is True, the low level performs its H_l steps in the environment trying to get to the current goal (lines 19-29). During the run, we store every step done with -1 reward in the low level ER (line 23). If we happen to achieve the goal (line 22) we pop out the current goal from `goal_stack` and break the low run (lines 25-28).

After the low level run we save current state to the solution (high level steps) (line 30). If the `goal_stack` is empty it means that the ultimate episode goal was achieved, the episode is finished with success (lines 31-33).

After every episode we roll out the solution to the vicinity, and then we create the tran-

Algorithm 4 Reverse Curriculum Vicinity Learning (RCVL)

Require: `high_agent`, `low_agent`, `is_reachable(s,g)` function, H_l low horizon, H_h high horizon, r radius for collection.

```
1: preliminary_goal_acquisition() - algorithm 7
2: for episode=1:num_episodes do
3:     num_subgoals = 0
4:     $s \leftarrow$ initial episode state
5:     $g \leftarrow$ environment_goal
6:     solution $\leftarrow$ [s]
7:     goal_stack $\leftarrow$ [g]
8:     while True do
9:         $g \leftarrow$ goal_stack[-1]
10:        reachable $\leftarrow$ is_reachable($s, g$) or exploration
11:        if not reachable then
12:            if num_subgoals $> H_h$ then
13:                Break. episode ends.
14:            end if
15:            new_ss $\leftarrow$ high.subgoal_suggestion($s, g$) - algorithm 6
16:            num_subgoals += 1
17:            goal_stack.append(new_ss)
18:        else
19:            for low_steps = 1:$H_l$ do
20:                $a \leftarrow$ low.select_action(s, g)
21:                Apply action a and observe next state s'
22:                achieved = (s' == g)
23:                store in low.experience_replay the tuple (s, a, achieved-1, s')
24:                $s \leftarrow s'$
25:                if achieved then
26:                    Remove g from goal_stack
27:                    Break for loop
28:                end if
29:            end for
30:            solution.append(s) and high.store(first state and last state of low run).
31:            if len(goal_stack) == 0 then
32:                episode is completed, break
33:            end if
34:        end if
35:    end while
36:    high.solution_to_vicinity(solution, r) - algorithm 5
37:    HER on low policy and high policy
38:    Every n episodes: Update networks
39: end for
```

sitions with HER. Every n episodes we update all the networks (lines 36-38).

4.5.3 Algorithm main components

- **Vicinity acquisition**
  The vicinity acquisition is performed during the preliminary acquisition, if done, and during the concurrent learning of the high level and low level. For the vicinity acquisition during the concurrent learning we collect the "high level steps" that are visited during the episode in `solution` list (lines 6 and 30 in the complete algorithm 4), and then we roll it out to store the data in `goal_list`. More details regarding `goal_list` structure are in appendix A. By "high level steps" we refer to the states that are visited every completion of `low_horizon` steps. When rolling out this `solution` (solution_to_vicinity() in line 36 in the complete algorithm 4, see algorithm 5), we have several settings to take into consideration.

  First, the **Radius of vicinity**: how distant the collected states from the sampled goal in terms of `low horizons`. It can vary between one low horizon, meaning the distance is feasible within one `low horizon`, to several horizons, when the maximum chosen was 6.

  Another decision that needed to be investigated was whether to store each trajectory in the order it was collected or **symmetrically**. For example if the agent went from state s_1 `low horizon` of steps till state s_2, we store that s_1 is a possible subgoal on the way to s_2, but should we also store that s_2 can be a possible subgoal on the way to s_1.

Algorithm 5 solution_to_vicinity

```
Require: high_agent, solution, r radius for collection, symmetry_flag
 1: solution.reverse()
 2: for i = 1:len(solution) do
 3:     for j=i+1:i+r+1 do
 4:         index = convert 2 dimensional solution[i] to 1 dimensional index
 5:         high/goal_list[index].add(solution[j])
 6:         if symmetry_flag then index = convert 2 dimensional solution[j] to 1 dimensional
    index
 7:             high/goal_list[index].add(solution[i])
 8:         end if
 9:     end for
10: end for
```

- **Subgoal suggestion**
  When the high level suggests a goal (line 14, see algorithm 6), it uses the value function approximation to calculate the possible values of the different possible paths. Every path is composed of two parts: one is from current state to the current subgoal, and second is from the subgoal to the next subgoal/goal. To visualise an example of this

please refer to figure 4.3. We can see the steps how the algorithm finds the more rewarding path to go through. This example is for a simple case, where obstacles do not exist, and the distance goes with opposite relation with the reward: the shorter it is, the more rewarding it is. For example in 4.3(a) we have 5 optional subgoals from `goal_list` for the current goal, and we calculate all of the following:

– $Q(path_1) = Q(s, sg_1, g)$
– $Q(path_2) = Q(s, sg_2, g)$
– $Q(path_3) = Q(s, sg_3, g)$
– $Q(path_4) = Q(s, sg_4, g)$
– $Q(path_5) = Q(s, sg_5, g)$

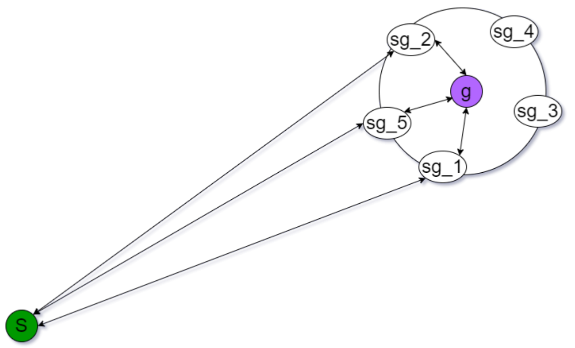
The suggested subgoal will be the one that draws the path with the maximum value estimation. In this case, if the value function is close to the real value, we expect the subgoal sg_5 to be selected.

This process is done recursively until a suggested subgoal is reachable from the current state and the low agent starts to act. it should cross the way that is presented in 4.3(e). If in any of the intermediate subgoals, again one is not reachable, then another suggestion from the high level can be made in the same way, until the low subgoal is reachable from the current state.

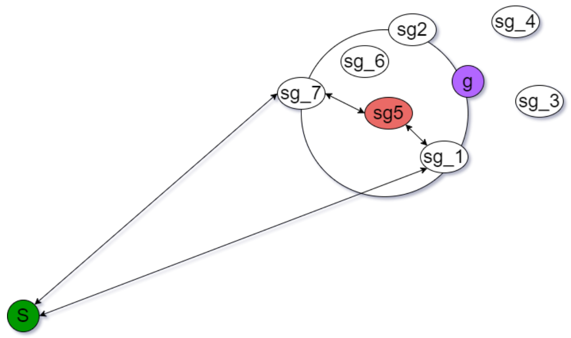
Algorithm 6 subgoal_suggestion

```
Require: high_agent, $g$ current target, $s$ current state.
1: index = convert 2 dimensional goal to 1 dimensional index
2: Possible_actions $\leftarrow$ high_agent.goal_list[index]
3: Broadcast $s$ and $g$ to dimensions of Possible_actions
4: Calculate Q_value($s_{broadcast}, Possible\_actions, g_{broadcast}$)
5: Choose the action that maximizes Q_value
```

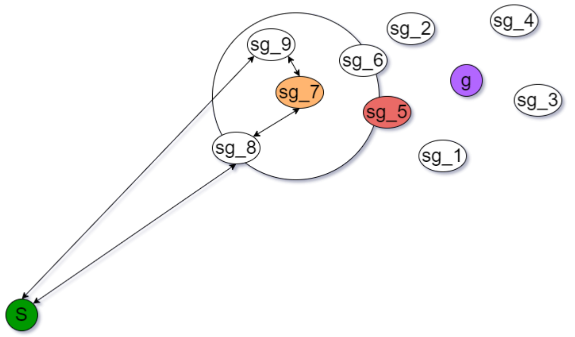
- **Preliminary vicinity acquisition**.
  In the case where we perform the vicinity collection also as a separated step, see algorithm 7, in practice we perform some goal sampling before we start the concurrent learning. In this phase of the algorithm we uniformly sample the goal space (which is the same as the state space). For each sample we initialise the environment, receiving a state and a goal. We perform primitive steps using the ϵ-greedy low policy trying to reach from the state to the goal. We perform a `low horizon` of steps and then set the original goal to be the state. That means that we go approximately to the same direction we came from, but not exactly since we also explore with probability of ϵ, see figure 4.4 (this technique is compared to a flatten acquisition in section 5.2.3). We store the first state, and then we store the visited state every `low horizon` steps. That



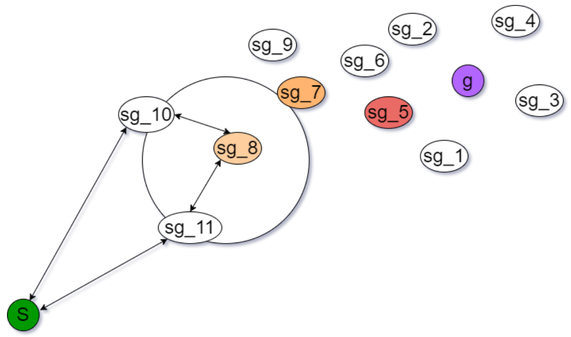
(a) Step 1: Evaluate the benefit (value approximation) of passing from state s to goal g through the possible subgoals $\{sg_1, sg_2, sg_3, sg_4, sg_5\}$.



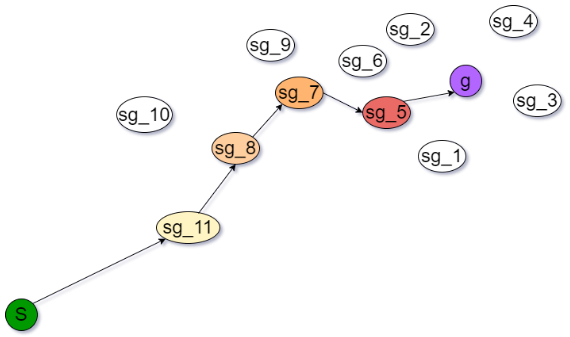
(b) Step 2: The path through subgoal sg_5 is the most rewarding, so it is now set as the goal, and the same process of evaluating the subgoals around it are done.



(c) Step 3: Subgoal sg_7 is detected as the most rewarding subgoal among the possibilities. The same process is repeated for the vicinity of sg_7.



(d) Step 4: Subgoal sg_8 is detected as the most rewarding subgoal among the possibilities. The same process is repeated for the vicinity of sg_8



(e) Step 5: Finally subgoal sg_{11} is reachable from the current state s, and we can see the revealed path that the algorithm suggested to cross the way from s to g.

Figure 4.3: A simple example of subgoal suggestions from learned vicinity of each of the goals/subgoals. The path is expected to get better as the value function approximator gets closer to the real value function, and as the agent learns the neighborhoods better.

is the equal to a single "step" of the high policy. That is the same as in the case of the concurrent learning phase acquisition.

This phase also integrates low policy training, which was decided after experimenting the consequences of it on the algorithm learning. Details regarding the this experimentation is brought under section 5.2.2.

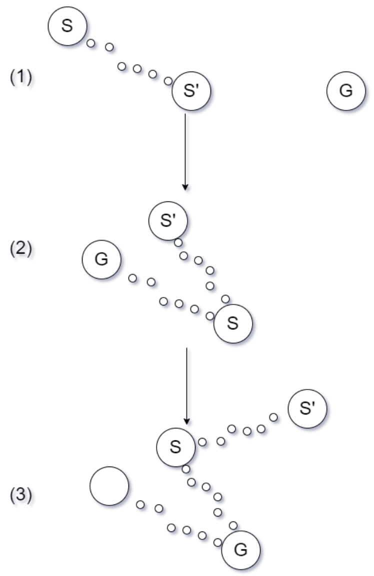


Figure 4.4: Preliminary vicinity acquisition with "back and forth" technique. (1) First we have state s and goal g, and the low agent tries to reach the goal applying ϵ-greedy policy. It ends a horizon of primitive actions in state s'. (2) Now the original state is set to be the goal g, while s' becomes s. g is no longer important, and we keep on sampling around the states that we are. Having ϵ-greedy policy, after a low horizon of primitive goals, the agent ends up in s'. (3) Again the process repeats itself, and we end up in a new but relatively close state.

The vicinity has the following possible properties that need to be tuned and set, in addition to the radius of the vicinity and symmetrical acquisition explained above:

1. **Sample size**: How many times to initialise the environment and perform the process of acquisition.
2. **Back and forward or flatten collection**: The agents performs `low horizon` steps then the goal can be updated to be the initial one, so the agent is expected to back, however, not exactly, since we use the ϵ-greedy policy of the low agent. Another option, instead is to leave the goal the same for all the repetitions, so that we perform mini trajectories on the way to the same goal. For visualisation please refer to 4.4.
3. **Number of repetitions**: how many of the mini-trajectories described above (back and forth or "flattened" path) the agent performs.

Algorithm 7 preliminary_vicinity_acquisition

```
Require: low_agent, high_agent n_samples, n_repetitions, flatten_flag, $H_l$, r radius for collection
1: for i = 1:n_samples do
2:     $s, g \leftarrow$ environment initialization solution.append(s)
3:     for j=1:n_repetitions do last_state = run_low_steps(low_agent, $h_l$, s, g)
4:         if Not flatten_flag then:
5:             $g \leftarrow last\_state$
6:         end if
7:         s = last_state
8:         solution.append(s)
9:         HER on low agent accumulated states.
10:    end for
11:    high_agent.solution_to_vicinity(solution, r) - algorithm 5
12: end for
```

4.5.4 Learning Recursive Goal Proposal by value estimation

We have also investigated the option of using the value approximations of the network to be stored in the ER as reward as explained in the second approach for long-term reward estimation in 4.3.3. The process of collecting and learning the neighbourhoods is done in a similar way to the other approach, and also in the way of suggesting subgoals.

For suggesting subgoals (action selection of the high policy) the algorithm goes through all the possible states that are in the list corresponding to the current goal, and calculates which is the most rewarding path to go from state s to goal g through. The one that has the maximum value (or minimum cost) will be chosen as a subgoal, and will be suggested as the next goal for the low policy.

The major difference of storing the estimations in the ER as explained in the seconds approach, makes the learning way more complicated, since there is nothing empirical that is fed into the reward estimations. In the beginning of the learning, the suggested subgoals are expected to be random, however with time, assuming the value estimators are converging to their true value, they are expected to be relevant. After several experiments, we reasoned that maybe there is no learning since the random initialization results with low estimation for the value. We decided then to test several different **initializations of the last value NN layer bias** that were expected to encourage learning. That also resulted in failure to learn.

The possible reason for the failure is that we have combined three characteristics: Function approximation, bootstrapping and off-policy learning. Those three properties integrated together might lead to divergence of the algorithm.

Chapter 5

Experiments and Results

The most relevant experiments performed in this work will be presented in this section. First the environments used will be presented, then some preliminary experiments, and finally performance comparison with State of the Art and Baseline. The preliminary studies that are documented are the ones that are related to the `RCVL` algorithm, the successful one. Similar studies for hyperparameters were done also with `RCRL` however when it did not demonstrate proper learning, it became redundant to document and conclude regarding it. Even though, in the final comparison, the performance of the two algorithms is presented and discussed.

The implementation details for both algorithms appear in appendix A. The default values are as appears in the appendix, unless said otherwise. In the preliminary studies and experiments, the default values are used, and only a single variable, currently characterised parameter, changes.

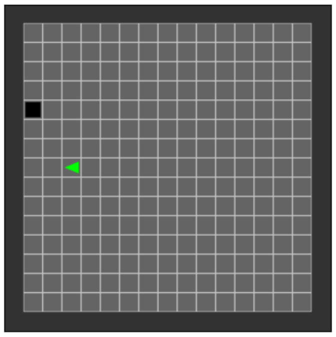
5.1 Environments

5.1.1 Minigrid Empty room

Minigrid empty room [31] is a mxn grid table, which is a simplified version of `Gym MiniGrid` [32] as appears in figure 5.1. The green arrow represents the agent while the black tile is the episode goal. When new starting states/subgoal is suggested they appear in different red - orange gradient as appears in figure 5.1(b).

The `state space` contains all the position on the board plus a direction: (x,y,dir). The possible values for x are 0 to m-1 and for y are 0 to n-1 according to grid width and height. The possible values for the directions are 0-3: right, down left and up. The `action space` contains three actions: turn left (0), turn right (1), and move forward (2).

The original goal space had only (x,y) location property. As explained in the previous chapter, the algorithm proposed also learns the directions for goal states. In that way the learning is more specific, and the `state space`, `goal space` are the same as the high policy `action space`. The mapping from state to goal is not longer needed.


(a) Empty room, target and goal visualization

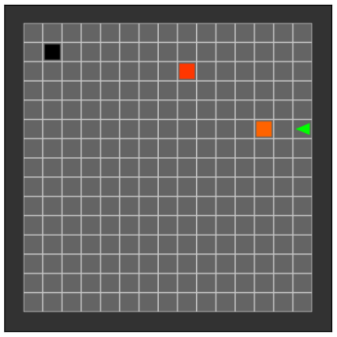

(b) Empty room: state, target with subgoals

Figure 5.1: Empty room example

To allow implementing algorithm of the `Hierarchical Reverse Curriculum Learning` 4.4, the ability of positioning the agent in a desired location was needed to meet the first assumption that in that paper [28].

The `reward function` is sparse, meaning only when the agent reaches the episode goal, it receives 0, and otherwise it receives -1 for every step.

Let up express the above formally:

- $\mathcal{S} = \{0, 1, 2..., m-1\} \times \{0, 1, 2..., n-1\} \times$ {Right,Down,Left,Up}.
- $\mathcal{G} = \{0, 1, 2..., m-1\} \times \{0, 1, 2..., n-1\} \times$ {Right, Down, Left, Up}.
- $\mathcal{A} =$ {turn left, turn right, step forward}.
- $\mathcal{R} = \{-1, 0\}$

5.1.2 Four Rooms MiniGrid

The environment is built of mxn grid table, as in the empty room, but it is divided by walls into four rooms, and it is possible to move from one room to an adjacent room via one tile [32]. Each tile (state) in this grid world might be either free, meaning the agent can be in it, or it is a wall, meaning it is an impossible state for the agent. That means that the agent needs to learn to align to this "key" tiles, and perform a "forward" action to move to another room. An example of environment with 15x15 tiles can be visualised in Figure 5.2:

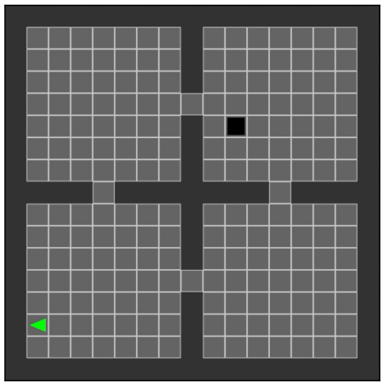


Figure 5.2: 15x15 Minigrid FourRooms

5.2 Studies During Development - configurations

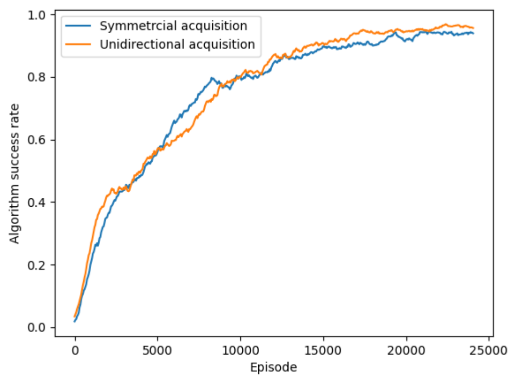
In the following subsections we go through different properties of the learning that were needed to be characterized for development. In order to be adapted to the complex environment we want to solve, the studies and experiments were made with the **Four Rooms Environment**. In addition, different properties/hyper parameters choice might result differently. But for simplicity, and to isolate the independent variables, only one configuration was chosen for use in all of the experiments. This can be found in Appendix A.

5.2.1 Symmetry in the vicinity acquisition

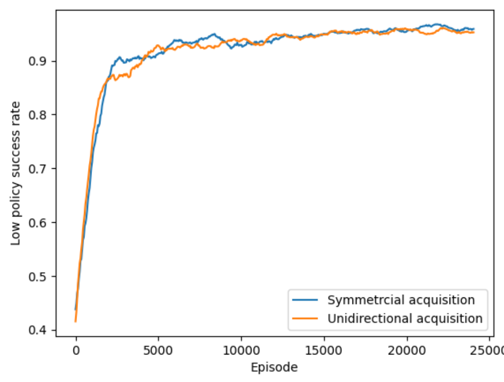
The algorithm `Reverse Curriculum Vicinity Learning` 4.5.4 creates during the the learning a set of neighbors for each goal (goal_list), from where it is reachable. This list could be saved straightforward in the way the agent visits the states, and in that way all the possible paths suggested by the `high policy` are paths that the agent had visited in the very same order. On the other hand, having our environment a reversible one, when a path from state s_1 to state s_2 is possible, means that the other way around is also true. That is, the path from state s_2 to state s_1 is also possible. For the `high policy` larger `goal_list` means more information that it might exploit for subgoals suggestions. Therefore we get double information for the high policy with the same amount of steps in the environment with the symmetrical acquisition.

To characterize the behavior of the policy with these two approaches of symmetrical information storing, we perform several experiments where all of the hyper parameters are the same except for the symmetrical states collection. In figure 5.3 it is possible to visualize the effect of the symmetrical goal storing on the performance evolution during training. For this test we used the algorithm with no preliminary acquisition. Other test have been performed with preliminary acquisition and showed similar tendency. Figure 5.4 presents the number of subgoals proposed during learning for both cases of collection.

We see that the symmetrical collection does not contribute to the algorithm performance, although enlarges the `goal_list` in doubled rhythm. It does not result with higher success



(a) Algorithm success rate: Goal storing - Symmetrical Vs. unidirectional



(b) Low level success rate: Goal storing - Symmetrical Vs. unidirectional

Figure 5.3: Minigrid FourRooms 15x15: Symmetry collection effect on success rate during learning.

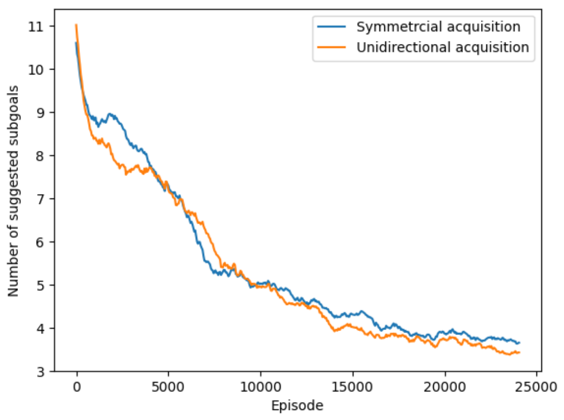


Figure 5.4: Minigrid FourRooms 15x15: Symmetry collection effect on number of subgoals suggested during learning.

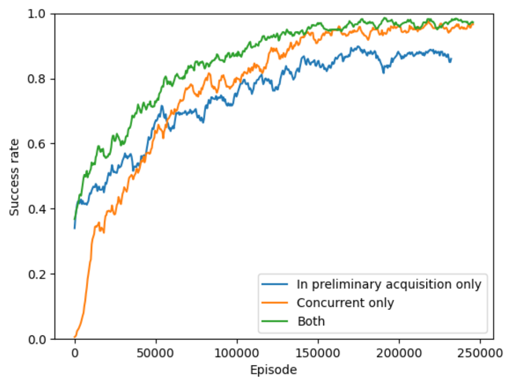
rate, but the opposite. In addition, the number of subgoals proposed, which we hope to get as low as possible (as long as the success rate is not defected), is higher for the case of symmetrical collection. Recall that more subgoals with the same success rate means **inefficiency**. Usually it means that more lower level steps are needed, and that the created path by the higher level is suboptimal. More subgoals proposed might point that more subgoals are not reachable to the lower lever, which makes sense in the symmetrical acquisition case, having stored goals in `goal_list` but not in `is_reachable`. Or in case of exploration, the low level has not mastered those trajectories yet. Hence the default for this property is chosen to be **unidirectional** collection only.

5.2.2 Low Policy learning phase

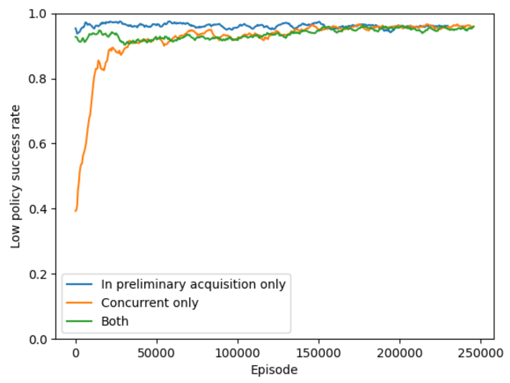
Having the first phase of preliminary acquisition, several different configurations are optional for when to train the low policy as brought in table 5.1. The learning that both high level and low level perform in the same time is referred as **concurrent learning**. A series of experiments were performed to compare the performance of each configuration. The representative results, for a reasonable amount of preliminary acquisition of 5,000 sampling size, were selected and presented in figure 5.5. We can see that combining the low level learning during the preliminary acquisition contributes a bit to the algorithm performance. The success rate when skipping concurrent learning is the worst. This is expressed both in the success rate of the algorithm and the number of subgoals proposed. Interpreting them together: skipping concurrent learning results with more subgoals suggestion and less successful episodes. That can be explained by the fact that the preliminary acquisition is characterized by a particular way of neighborhood collection as explained under section 4.5. This learning helps the low agent to be more successful but during the concurrent learning it has different tasks, so learning adaptation to the different task is needed. Summing this up, **the learning of the low level should be integrated during preliminary acquisition, if done, and always during the concurrent learning with the high level**.

	Preliminary acquisition	**Concurrent learning**
Low policy Learning phase configuration	X	
		X
	X	X

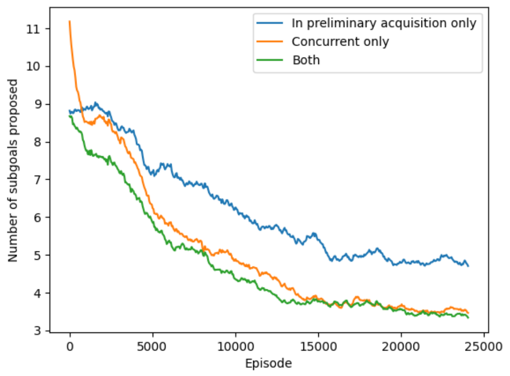
Table 5.1: Possible configurations for low policy learning phase



(a) Algorithm success rate



(b) Low policy success rate

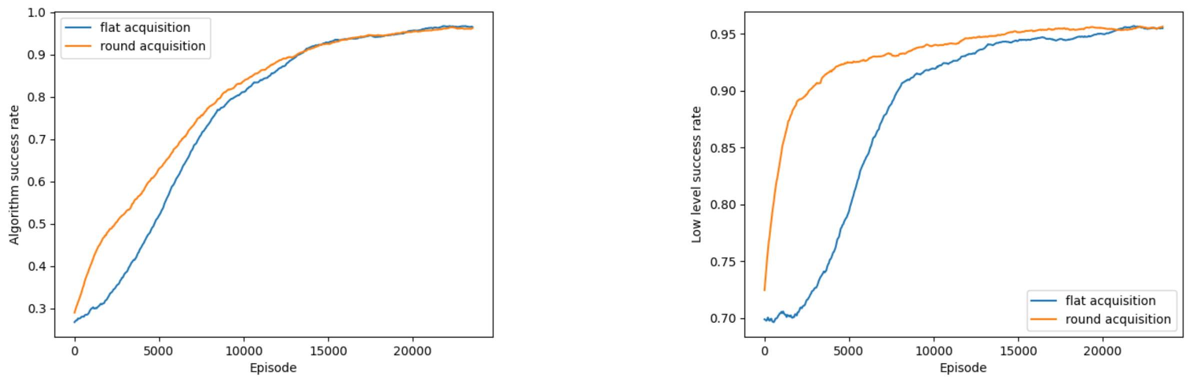


(c) Number of subgoals proposal comparison between the three possible configurations.

Figure 5.5: Minigrid FourRooms 15x15: Low policy training configuration comparison

5.2.3 Flat acquisition Vs. round (back and forth) vicinity

For the vicinity acquisition, one way of doing it was presented in section 4.5.3, which is the ”round”/”back-forth” technique that is eventually used in the final version of the algorithm. However, it is possible to collect also in a ”flat” way. That means that after a low agent performs low horizon primitive actions, we store the solution, but continue with another horizon towards (approximately) the same goal. It is similar to the case of the vicinity during the concurrent learning, but in this case there is no change in the goal (subgoal suggestion), and all of those primitive actions are done consequently by current ϵ-greedy low policy. Figure 5.6 shows the suggested ”back and forth” technique contributes to the performance with a reasonable 5,000 samples in preliminary acquisition. Observing the low level success rate, it seems that learning and acquiring in the ”round” manner helps the low agent to master the learned trajectories, having the success rate so high from the beginning of the concurrent learning. Eventually this gap is closed and both ways converge to similar success rates.



(a) Algorithm success rate (b) Low policy success rate

Figure 5.6: Minigrid FourRooms 15x15: Flatten acquisition Vs. round acquisition (back and forth).

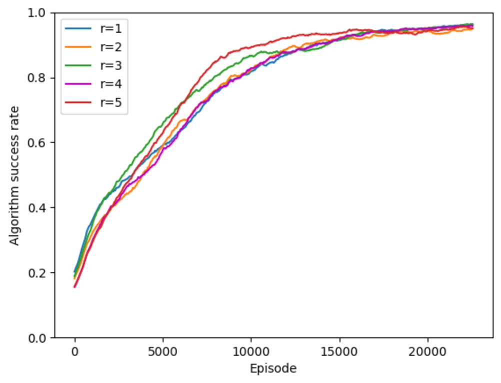
5.3 Preliminary experiments - Hyperparameters search

Similar to previous section, in the current section all of the experiments are done in Minigrid FourRooms environments. Unless claimed otherwise, the implementation details are as appears in appendix A.2.

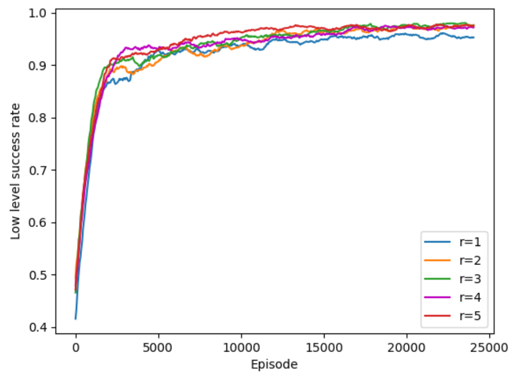
5.3.1 Radius for states collection

As we saw in algorithm 5 we need to set the radius, for vicinity collection we need to set a radius. The radius defines how large the vicinity is around a goal. **It is measured with the "unites" of `low horizons`**. For example when the radius equals two, the neighbors we collect into `goal_list` can be located up to 2 low horizons away from the goal. Therefor, we need to understand which radius value would result with the optimal solution. In addition, we would want to suggest subgoals that are not necessarily proximal to the goal state, to be able to suggest less subgoals. In figure 5.7 we can see a characterization of this effect. It could seem like the larger radius show better performance in the beginning of the learning process. But then, as the learning proceeds, we see a convergence to similar success rates for all the different radii. Table 5.2 presents the average number of steps per episode, calculated across 1,000 episodes with random environment initialization. we see that the lower radius results with fewer steps. That means the path built by the high level is more optimal: it reaches the same success rate with fewer steps. The difference is not significant (less than 1 step difference), however enlarging the radius adds run-time complexity, and do not contribute to the success rate in the long run, neither to the path optimality, meaning it is better resolution to stick to radius of 1 low horizon.

A possible reason for the difference in low level performance for the worse for radius of 1, is that the accumulated neighbourhoods are more diluted, that might lead to subgoal suggestions that are in the limit of the knowledge of the lower level. It might be that states that are in the list of `is_reachable`, but have not yet been mastered by the policy itself. The results are that supposedly reachable subgoals within one horizon are suggested but the lower level is not well mastered. It is possible that with higher radius, the vicinity contains more possible states to suggest and it is able to choose the better cheapest path. However, the bottom line is that with **radius of 1 we get the same algorithm success rate, less complexity and similar path optimality**.



(a) Algorithm success rate: vicinity radius effect



(b) Low level success rate: vicinity radius effect

Figure 5.7: Minigrid FourRooms 15x15: The effect of different radius in collection. Each unit of the radius equals one `low horizon` primitive steps.

Radius [units of low horizon]	**Average steps per episode**
1	**16.83**
2	17.52
3	17.31
4	17.24
5	17.17

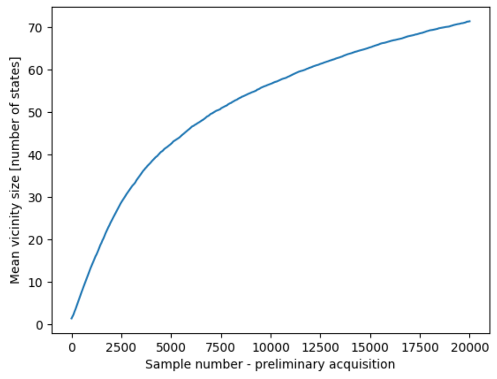
Table 5.2: Minigrid FourRooms 15x15: Average steps per episode across 1,000 random environment initialization.

5.3.2 Samples size in preliminary acquisition

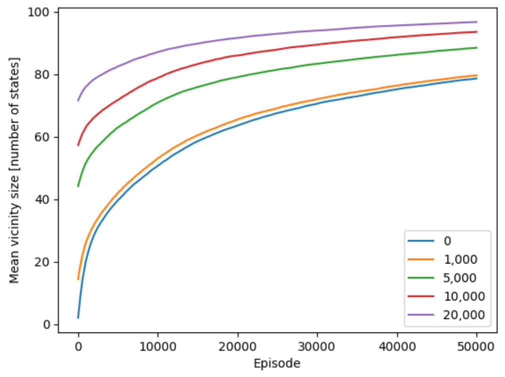
The preliminary vicinity learning, as explained, is done before the algorithm is trained as a whole (concurrent learning of high policy and low policy). As the preliminary acquisition phase is longer, the vicinity collection naturally increases. Having `goal_list` (learned vicinity for each state) the only storage where the high policy can suggest subgaols from, it was essential to track its evolution along the learning. The size of the vicinity is measured by the mean number of states accumulated for each state across all states. It grows during the preliminary acquisition phase, if done, and during the concurrent learning phase.

Figure 5.8a presents the mean accumulated number of states in the vicinity along with the preliminary acquisition. It is possible to see that the accumulation is very fast in the beginning and becomes slower, however it stays relatively significant all along the acquisition. Then the effect of different amount of preliminary acquisitions (samples of goals) on the accumulation of vicinity size during the concurrent learning is presented in figure 5.8b. It is clearly seen that the more preliminary acquisition is done, the knowledge about the vicinity is larger (more states are collected for each goal). This is further emphasized in the mean final neighborhood size in table 5.3. The concurrent learning does not compensate for the vicinity size by the end of the concurrent learning. However the rhythm (**mean derivative function** of the accumulation function) of accumulation is greater as the preliminary learning is smaller: 0.035 and 0.015 respectively, which is 60% slower. That makes sense with same reason that the rhythm of accumulation during the preliminary acquisition gets smaller as more knowledge was already collected. The possible reason is that we accumulate already known neighbors, so they contribute less to enlarge the known neighborhood. In addition the lower level is more successful, and less explores, then the collected states are already known, not new.

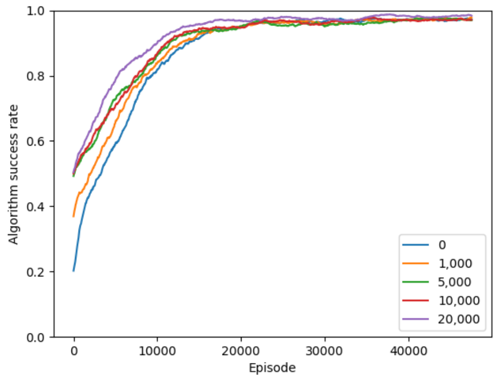
The effect of the sampling size during preliminary acquisition on the algorithm success rate is presented in 5.8c. We can see that **asymptotically the different setups converge to similar success rate, although the preliminary acquisition shows its contribution in accelerating the learning**, reaching to better success rates faster. For further inspection the low level success rate is presented in 5.8d. Having the low level more successful of course enables the performance of the algorithm as a whole. It does not assure the whole algorithm success, but without its success, it is impossible the algorithm as a whole will be successful. It is an essential condition, but not sufficient. Also, for the low level success, the different setups converge to similar success rates. Further discussion regarding the trends in low policy success rate for this algorithm will be discussed under section 6.2.1.



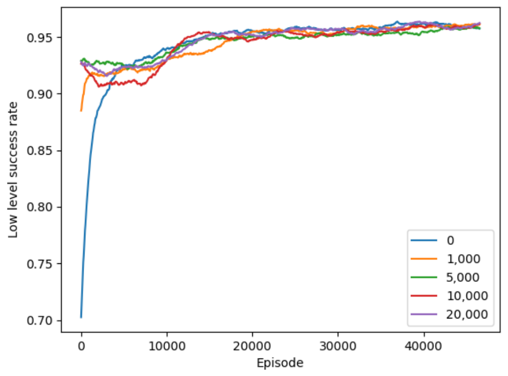
(a) Vicinity size in preliminary acquisition.



(b) Vicinity size during the concurrent learning, having different sampling size in preliminary acquisition.



(c) Algorithm success rate according the sampling size in preliminary acquisition.



(d) Low level success rate according the sampling size in preliminary acquisition.

Figure 5.8: Minigrid FourRooms 15x15: Vicinity size (in terms of number of states) characterization along learning when setting different number of goals to sample during preliminary acquisition.

Sampling Number in preliminary acquisition	**Final mean vicinity size [number of neighbors]**	**Mean accumulation rhythm**
0	78.7	0.035
1,000	79.7	0.0349
5,000	88.515	0.02
10,000	93.61	0.0199
20,000	96.77	0.015

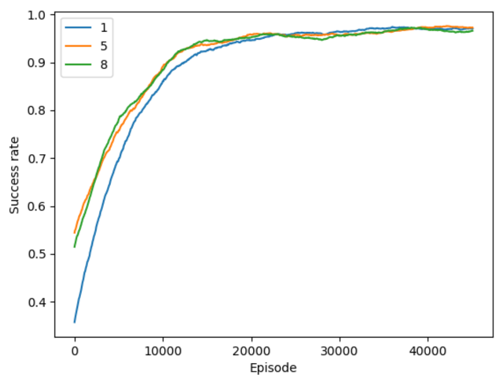
Table 5.3: Neighbors accumulation with different preliminary acquisition sampling number.

5.3.2.1 Runtime implication of preliminary acquisition

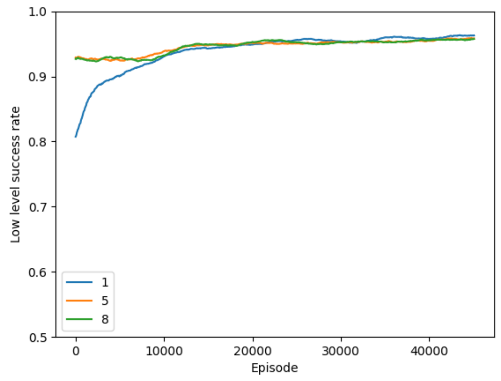
The preliminary acquisition in is practice performing interaction with the environment, which might diverse in the time addition when changing its parameters such as `low horizon`, number of repetitions, number of goals to sample and so on. Calculating on sampling size of **20,000** in the preliminary acquisition with several different parameters, the addition time required for the preliminary acquisition **ranges between 4% to 7%** from the concurrent time learning, depending on the acquisition hyper parameters. This is not a big addition given that those are only short sampling rounds of mini-trajectories (not as in the concurrent learning, where we have in each episode `high horizon` of mini-trajectories), and there is no large complexity while doing that.

5.3.3 Back and forth while preliminary acquisition: number of repetitions

When acquiring the samples of the vicinity, as explained, after the agents performs horizon primitive actions, we try to make the agent to repeat its steps back with the ϵ-greedy policy. The number of repetitions on that process might affect on both the learning phases of the low policy (assuming low policy training during goal acquisition and during concurrent learning). We have performed a study to characterize the effect of repetition number, executing the experiment with a reasonable sample size of 5,000 in the preliminary acquisition, and different number of repetitions ranging between 1 and 8. Representative results are shown in figure 5.9. The results are very similar for low policy. With one and only repetition low policy starts with lower success rate. In comparison, with 3 or 8 repetition it starts higher. However 8 repetitions do not seem to contribute much over 5 repetitions. As for the algorithm as a whole, 5 repetition contributed a bit, while 8 repetition results with similar success rate. A reasonable decision is then to stay **with 5 repetitions of back and forth during the preliminary acquisition**.



(a) Algorithm success rate

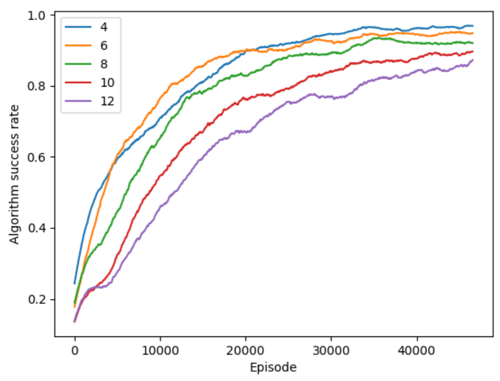


(b) Low level success rate

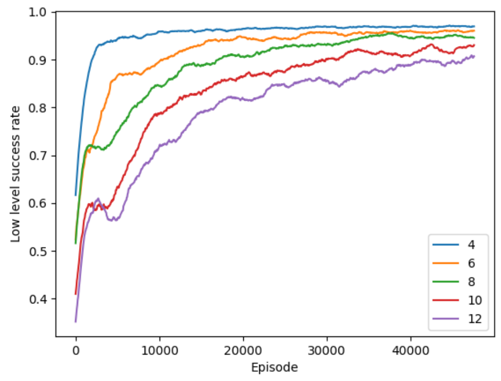
Figure 5.9: Minigrid FourRooms 15x15: Back and forth repetitions during preliminary goal acquisition.

5.3.4 Low level horizon effect

We have tested the effect of different low horizon on the training, and the results appear in figure 5.10. The most successful choice appears to be a **horizon of 4**. Horizon of 6 results with similar success rate as 4 for the algorithm as whole, but the low level success shows a clear advantage of 4 over 6. The low level success that increases very fast for the shorter low horizon, allows the high level to prosper accordingly. This is a very important parameter since as we see, the results do not converge to the same value even after 50,000 episodes.



(a) Algorithm success rate

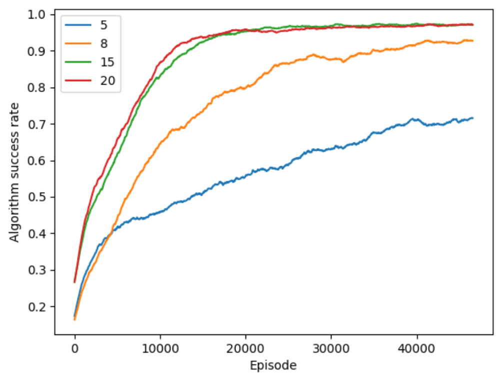


(b) Low level success rate

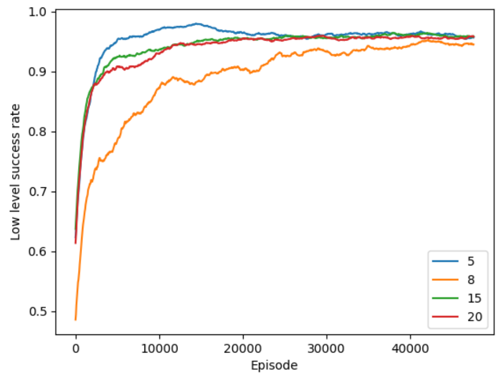
Figure 5.10: Minigrid FourRooms 15x15: `Low horizon` success rate effect on the learning

5.3.5 High level horizon effect

Another important hyperparameter for final behavior is the maximum number of subgoals that are allowed to be proposed during an episode, that is the high level horizon. The characterization of its effect is presented in figure 5.11. The idea is to choose a minimal number of subgoals, as long as it does not harm the performance. The reason is simple: less subgoals with the same low horizon means less primitive steps. If we converge with an increasing amount of subgoals to the same success rate approximately, it means that there are redundant subogals. That is why the better choice would be the minimum subgoals with maximum success rate. Observing the results it is clear that horizon of 5 is too low. Similarly horizon of 8 also seems suboptimal. It is sufficient for most cases but it fails to be enough in some setup of state and goal. That is why it does not converge to the same as the others. On the other hand 15 and 20 behave very similar , which implies we might see redundant subgoals for 20, that makes the agents path longer. **A good choice for the high horizon in that case would be then 15**.



(a) Algorithm success rate



(b) Low level success rate

Figure 5.11: Minigrid FourRooms 15x15: `High horizon` success rate effect on the learning

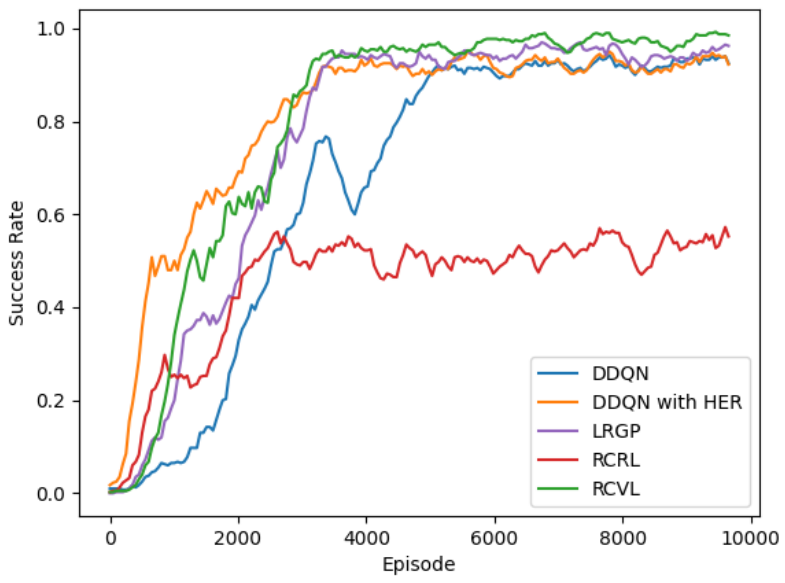
5.4 Comparison with State of the Art and baseline

The following sections present the comparison of the proposed algorithms with the baselines. As for RCVL, all the presented results are without **preliminary acquisition**, meaning this is the "worst" behavior with the algorithm. With the addition of preliminary acquisition the learning curves for RCVL are better. The rest of the parameters are brought in appendix A.

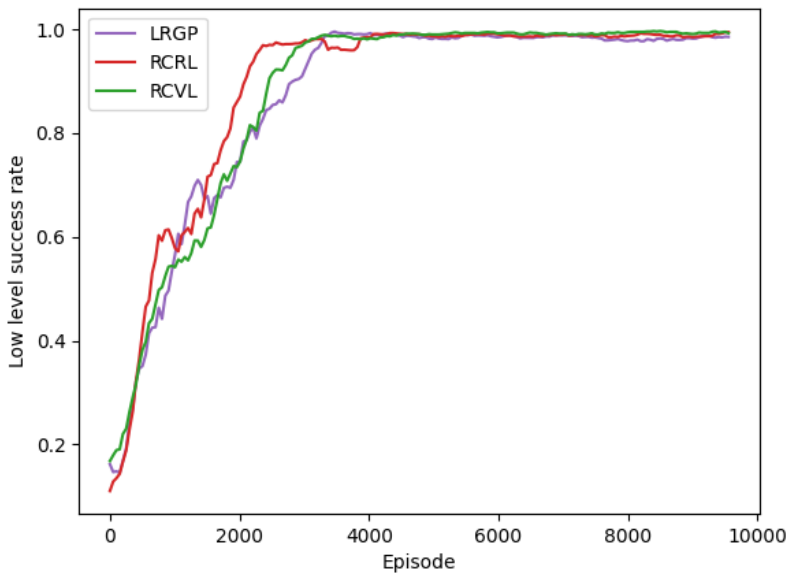
5.4.1 Minigrid empty room

It is possible to visualize the learning process of the proposed algorithms: RCRL, RCVL in figure 5.12a, in comparison with LRGP (formerly proposed baseline) and State of the Art algorithms: DDQN and DDQN combined with HER. Observing the success rate of the different algorithms, we can see that RCVL performs similar to the LRGP and to DDQN with HER, reaching close to 1. RCRL fails to learn well the environment, reaching around 0.5 success rate. As for the low level success, figure 5.12b, the three algorithms compared show similar learning curves.

To show the optimality of the paths drawn by the high level of RCVL, let us observe the average number of steps per episode. DDQN and its variation with HER learn the optimal behaviour (at least close to optimal) in the empty room (it was tested by observing it in action). Also, for the simpler environment we expect DDQN to master the tasks. That is, good performance of the tested algorithms will results with similar number of steps per episode. To evaluate this we ran 1,000 episodes with random initialization in the Minigrid Empty Room 15x15 environment, and calculated the average number of states across all of them for episode completion. Recall that the success rate of all the algorithms is close to 1, which makes the comparison valid. Otherwise unsuccessful episodes of one or another might bias the average, since the unsuccessful episodes are expected to be longer ones. Both scenarios of taking them into account or ignore them results with some bias. Please refer to table 5.4 to observe the results. We can see that the optimal (shortest) path is done by DDQN with an average of 12.72 steps. RCVL testing resulted in 13.76, which is 8.1% more. We can understand that on average RCVL results with a path that has one step that is redundant. This can be the result of hierarchical learning, which is not needed in such a simple environment. The fact that LRGP resulted with an even higher number of average steps (17.04 which are 34% more than DDQN), supports this reasoning. Overall it is safe to say that the solution of RCVL is close to optimal with 1 step difference.



(a) Algorithm success rate: learning process of the proposed algorithms compared to the baseline (LRGP) and the state of the art (DDQN and DDQN+HER).



(b) Low level success rate: learning process of the proposed algorithms compared to the baseline LRGP.

Figure 5.12: Minigrid Empty room 15x15: learning process of the proposed algorithms compared to the baseline LRGP and the state of the art: DDQN and DDQN+HER.

Algorithm	**Average steps per episode**
DDQN	**12.72**
DDQN + HER	12.86
LRGP	17.04
RCVL (ours)	13.76

Table 5.4: Minigrid Empty Room 15x15: Average number of subgoals per episode calculated across 1,000 episodes with random initialization.

5.4.2 Minigrid FourRooms environment

5.4.2.1 Success rates

In figure 5.13 It is possible to visualize the comparison of the proposed algorithms to the baseline `LRGP` and State of the Art algorithms DDQN, DDQN combined with HER.

Additionally, Some numerical results are presented in table 5.5. The presented mean success rates are calculated over 500 last episodes in the scope mentioned (last 500 episodes reaching to 25,000 episodes and so on). Then a 1,000 episodes with random initialization was performed, and the mean across all those 1,000 is presented in the table. `RCVL` outperformed the rest of the algorithms during learning and during test.

It is possible to see that the baseline algorithms struggle a lot solving the problem, having DDQN reaching no more than 0.23 of success, and DDQN with HER succeed a bit more reaching up to 44%, **while `RCVL` reaches close around 0.97 success rate**. `RCVL` outperforms `LRPG`, having it around 0.8 success. Observing the low level performance during learning shown in figure 5.14, `RCVL` expresses similar learning to the baseline. Thirdly, as presented under section 5.3.2, addition of **Preliminary Acquisition** could accelerate the low level learning, leading to even faster convergence of the algorithm.

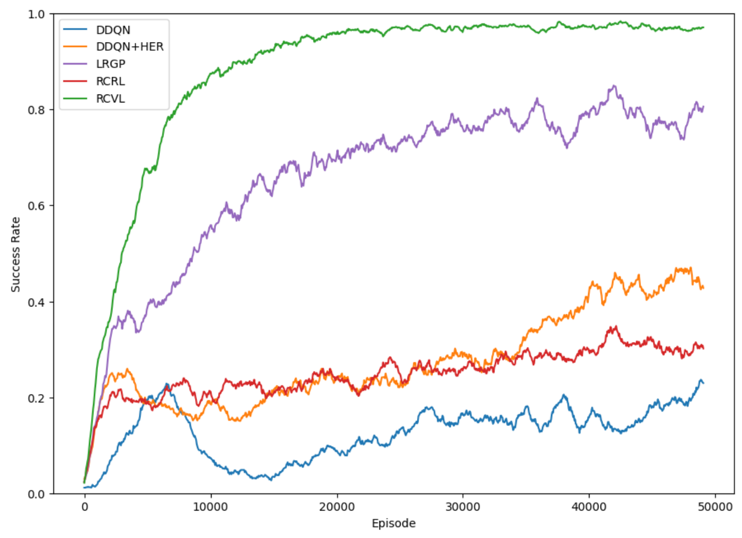


Figure 5.13: Minigrid Four rooms 15x15: Algorithm success rate comparison of the proposed algorithms with baseline `LRGP` and State of the Art: `DDQN` and `DDQN+HER`

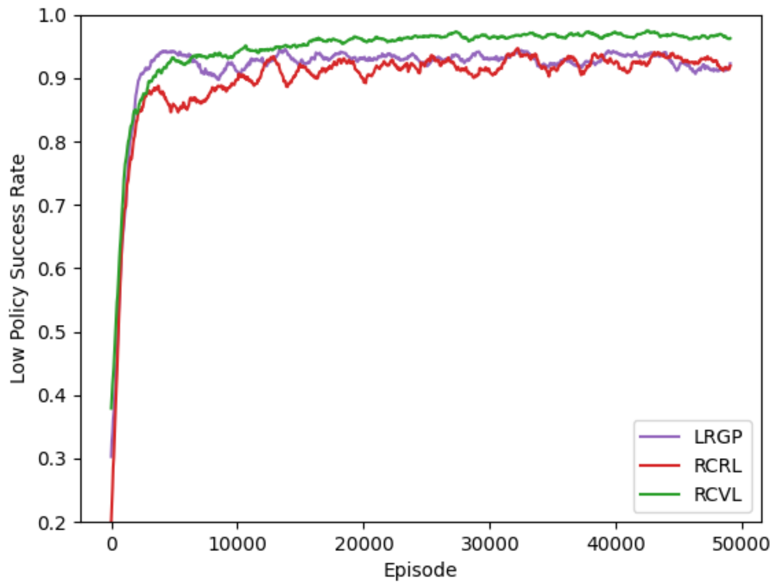


Figure 5.14: Minigrid Four rooms 15x15: Low policy success rate comparison of the proposed algorithms with baseline `LRGP` and State of the Art: `DDQN` and `DDQN+HER`

Algorithm	**Mean success rate after 25,000 episodes**	**Mean success rate after 50,000 episodes**	**success rate: Test over 1000 cases**
DDQN	0.1	0.22	0.23
DDQN + HER	0.22	0.44	0.44
LRGP	0.72	0.8	0.85
RCVL (ours)	**0.96**	**0.97**	**0.98**

Table 5.5: Minigrid FourRooms: Comparison of the suggested algorithm with baseline `LRGP` and State of the Art algorithms.

5.4.2.2 Subgoals proposals

For further comparison it is possible to observe the number of subgoals suggested along learning, having equal low `low horizon` of 4 (otherwise the suggested subgoals would not be representative for anything). Figure 5.15 presents the results. It is easy to see the the number of subgoals proposed is lower and steadier (smaller amplitudes) for the proposed algorithm `RCVL` over the baseline `LRGP`.

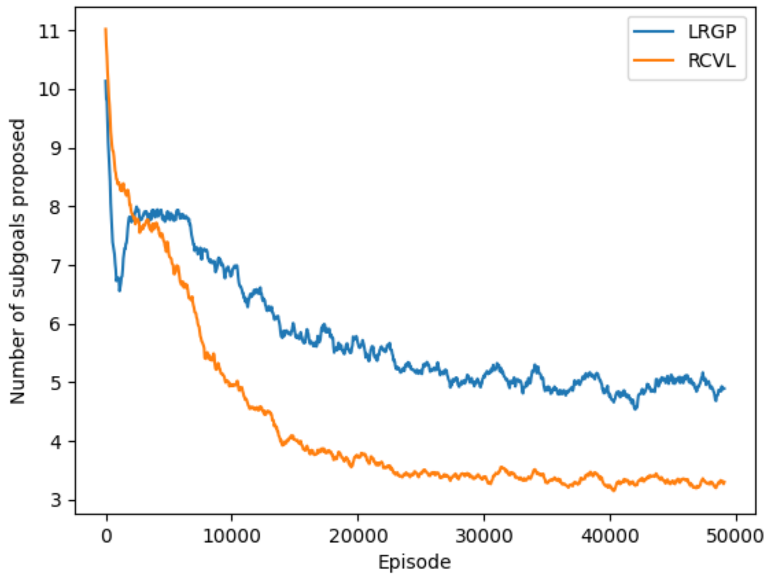


Figure 5.15: Minigrid FourRooms 15x15: Number of subogals proposal during learning for both algorithms `LRGP`, and `RCVL` (ours)

5.4.2.3 Path optimality

In the following examples in figure 5.20 it is possible to visualize the path drawn by the high level. The paths seem to be very direct and do not express redundancy.

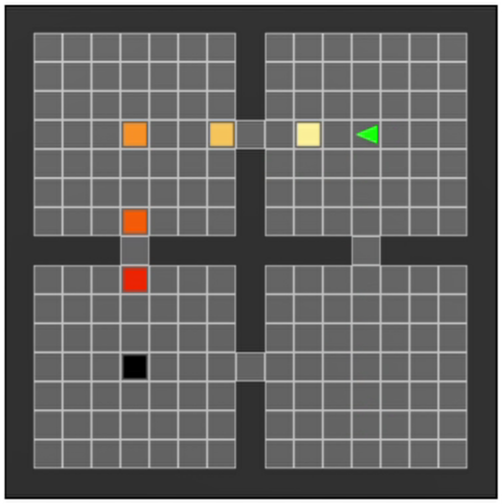


Figure 5.16: Drawn path by high level subgoals suggestions - example 1.

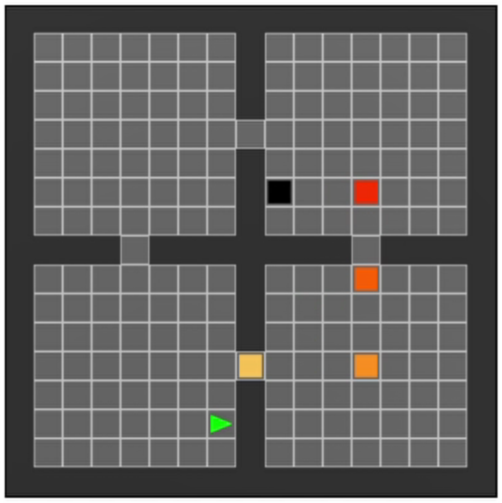


Figure 5.17: Drawn path by high level subgoals suggestions - example 2.

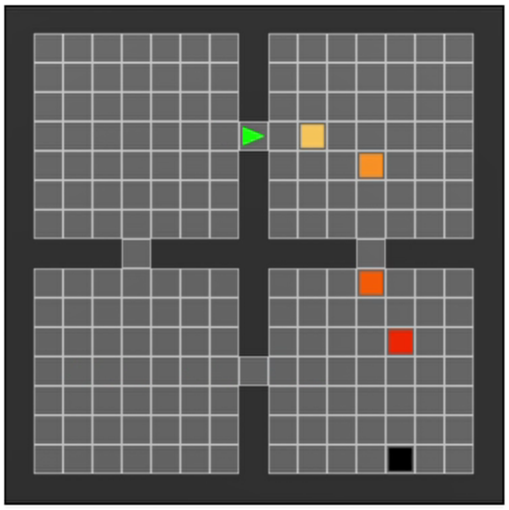


Figure 5.18: Drawn path by high level subgoals suggestions - example 3.

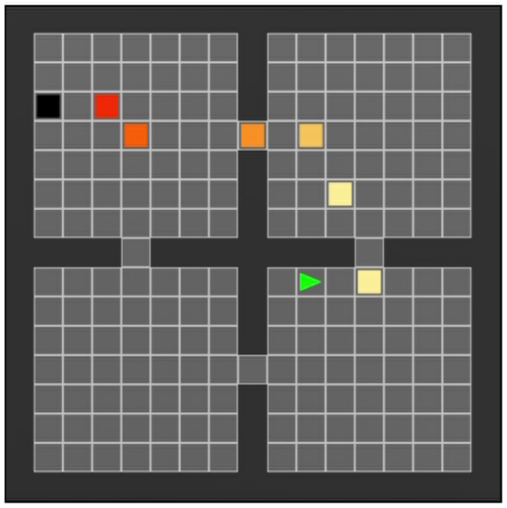


Figure 5.19: Drawn path by high level subgoals suggestions - example 4.

Figure 5.20: Minigrid FourRooms 15x15: Paths examples for visualization of the path drawn by the high level.

Chapter 6

Conclusions and Future Work

The **most successful algorithm** in the tested environments was proven to be **RCVL** where the results in the more complex environment of the Minigrid FourRooms showed a clear superiority over the state of the art and the `LRGP` baseline. In addition the problem of suggesting subgoals that are not feasible is solved with `RCVL`.

`RCRL` did not demonstrate ability to solve the tasks correctly. This could be related to several points of deficiency which will be suggested in section 6.1.1.

The following sections discuss the results, trying to reason and conclude, for each of the suggested algorithms separately: first `RCRL` and then `RCVL`. For each of the algorithms there is also a section for possible future suggestions for further development or improvement.

6.1 Reverse Curriculum Recursive Learning (RCRL)

6.1.1 Discussion and conclusions

The algorithm `RCRL` was not successful in its proposed starting states. It causes the agent to advance to bad locations, which results in a **low success rate** for the algorithm as a whole, while the low level policy, when observed separately, reaches a very high success rate.

One possible reason for the failure could be the fact that when suggesting a new starting state, a **transition rollout** is done (both for the low level and high level). It can be either after one or several mini trajectories done by the low policy, or after zero. It implies that the transitions accumulation for the high policy is very limited, since that after each new proposal it is rolled out (stored) and reinitialized. For `LRGP` and `RCVL` it is done only at the end of an episode, which allows accumulating at least several runs of the lower level for most cases. On the other hand, it is not possible to rollout the transitions only at the end of an episode for this algorithm, since the **transitions are not accumulated continuously**. When the high level proposes a starting state, then we update our agent state to be located in that position, meaning the continuity in accumulating ”high level steps” (and also primitive steps) is interrupted. The high level needs this information, since only an accumulation of at least 4 high level consequent steps can produce a high level significant information (s, a, s', g). It is much less likely for the high level to learn sufficiently to happen with the current algorithm flow.

Another possibility for the failure is similar, but from a different perspective - the **value estimation of SAC** makes use of `Bellman's equation` where the next state is used for error calculation. The problem is that the "next states" from the tuples (state, action, next state, reward, goal) that we store during the consequent mini trajectories, are different from the flow of the algorithm in practice. The high level next state after a suggestion will be the suggestion plus the low horizon steps performed from it towards the current goal.

Making the algorithm to work, it will still carry the problem of **unfeasible starting states suggestion** by the high policy, since we use the explicit representation of the SAC, which is based on the approximators only, and do not have a direct instructions of which specific goals to avoid. In that case the solution would be similar to the solution that was applied in `LRGP` baseline algorithm, where a list of feasible states is accumulated, and when the subgoal suggestion is not appropriate then it is eliminated. Another subgoal is then being requested with addition of noise, to suggest a close location that is allowed.

6.1.2 Future Work

A new way for high level **experience accumulation** should be considered. In the current implementation along with the transitions, we store the reward as minus the high level steps (with the Monte Carlo inspired approach) from state until goal. With the `RCRL` we can not know the actual number of high level steps required between a state to action (suggestion of the high level), only until we go back to have the desired starting state in the current sub-trajectory to be solved. Recall that this will occur when the recursivity is folded back, after the last mini-trajectory (that reaches the episode ultimate goal) is solved, and recursively the mini-trajectories are achieved until reaching the specific scope. The true challenge here would be to suggest a representative reward system that correlates with algorithm flow and that will allow actual learning.

One possible method for new experience accumulation can be to store the states visited with a **levelling index**, that will be the key to every sub-trajectory that is expected to be solved during the episode. It can be grasped as a **pyramid** where the first level is always the sub-trajectory that starts with the episode starting state, given by the environment. The accumulation for the first level starts, and then, when another starting state is suggested, we start to store it to the second level, and so on. Then when the recursivity is folding backwards, we track this index and go back to store the visited states after the already stored states in the specific level (index), when we try to solve the sub-trajectory linked to this index. This can bypass the incontinuity that is integrated within the algorithm. Then, with Hindsight experience replay relabeling it will contribute to the high level knowledge without a doubt more than the currently accumulated knowledge. We can make use the very same Monte Carlo inspired approach for the reward stored, just in the same way is it applied with the current algorithm.

An additional change that could be considered, is to apply some kind of "inverse" to Bellman's equation used in `SAC`, in the sense that the error should be calculated for the **reward of the sub-trajectory between the state to the action** (suggested starting state). This is different from the current implementation where the estimation and error calculation is on the reward between the next state to the goal. That is in light the difference in the flow, where the estimation should be done for the interval between the state and the action, since it is the mini-trajectory that is solved after the (action, next state, goal) accumulation, only when folding the recursivity backwards.

The main problem of the algorithm lies in its flow, where a new goal suggestion **interrupts the sequential accumulation** of high level steps. Hence, another optional direction for improvement can be to **avoid the manual agent positioning**, and perform the mini-trajectories in a sequential mode. However, it is hard to think of a way to implement this idea while preserving the reverse curriculum idea.

6.2 Reverse Curriculum Vicinity Learning (RCVL)

6.2.1 Discussion and conclusions

Regarding the construction of the algorithm, given the results in the preliminary characterizations presented, it is possible to notice some trends.

The algorithm does not seem to benefit from the **symmetrical acquisition** of the neighbourhoods, but instead it slows the learning a bit. A possible reason could be that during adding information to the `goal_list`, the low level learns the trials that it performs in the order they are performed. That means that if we stored a state, as if a specific goal is reachable from, but the low agent did not go through them in the specific order, then the agent might not yet know to move in the opposite direction. Although the `high policy` could benefit from that information, it would not necessarily be useful when expecting the `low policy` to perform this path. With that being said, by the end of the training, the success rates difference seems pretty redundant. The bottom line is that **unidirectional acquisition is slightly better**.

The preferable **radius** for collection is 1, by a small difference of path optimality. A possible reason for that is because having it set to 1 assures that the goal is reachable from a neighbour within maximum of one low horizon primitive steps, but possibly less. That happens due to the fact that the agent might move not in a "straight" direction (U shape for example). By "straight" the meaning is an optimal shortest path from s_t to $s_{t+low\_horizon}$, and not necessarily a straight line. Let us recall that for radius r, we collect states that are maximum r low-horizons ($r \times low\_horizon$) away from the goal state. The larger r is, the described phenomenon is amplified, and the actual distance gradually tends to be less than $r \times low\_horizon$. The consequences of this phenomenon is that proposed subgoals are in practice gradually less than a **complete number of low horizons**. The result is that the needed number of subgoals is the **ceiling round number** of the distance. Let us recall that we have several sub-trajectories. Having each one them solved by ceiling round number of low-horizons, does not allow optimality. That is, with larger radius, we have the tendency to suggest more subgoals, which invigorates path sub-optimality. It is important to say that **if both the high and the low level are faultless, extra subgoals suggestion does not lead to path suboptimality**. However, in the real world, they are not faultless, so it is better to avoid this situation of incomplete low horizon distance. In the real world - the more subgoals suggested unnecesarily: the more probable that the high policy will draw sub-optimal path (and also the low policy performs extra steps if not faultless).

Radius of 1 makes the "proposed path for solution" by the high level being more direct and simple, allowing breaking the trajectories more efficiently. Usually for radius of 1 the algorithm builds the whole path before the low level starts acting. Then when it starts acting, it usually completes the whole path in a row, with no additional subgoals proposals in between. Let us clarify the idea with an example. With r=2 we might suggest for one sub-trajectory a subgoal that is 1.2 low-horizons away, and for the consequent sub-trajectory a subgoal that is 1.3 low horizons away. The whole trajectory is 2.5 low horizons long (optimally speaking). This will be solved with extra 2 subgoals otherwise the low level can't reach 1.3 of its horizon, which is summed up to 5 subgoals (the original, two suggested and another two extra suggested to advance 1.2 and 1.3 low horizons) in the whole trajectory. With r=1, we would be able to break the trajectory differently: first sugboal 0.9 low horizon away, second sugboal 0.8 low horizon afterwards, and third 0.8 low horizon afterwards.We solved the same 2.5 low horizons long trajectory with 4 subgoals (the original goal and three suggested). Note that if both high policy and low policy are faultless, this extra subgoals does not harm the optimality. The conclusion is that radius of 1 might help to avoid recursively suggesting subgoals "in the middle" with less than low horizon steps between. Let us further clarify the idea.

The important and **significant learning phase of the low level is the concurrent learning**, since even when having the low level starting from a very high success rate, if we skip learning concurrently with the high level, the high level does adaptation to the current abilities of the low level by suggesting subgoals that are suboptimal, but in most cases achieves the ultimate goal. It is possible that a larger sampling size in preliminary acquisition could help the low level to dominate all of the space even better, which could allow the high level to suggest more optimal paths (without concurrent learning). The addition of learning during the preliminary acquisition shows a positive contribution to the algorithm learning, and the best performance was captured when having the low level learning both during the preliminary acquisition and during the concurrent learning. The reason for that is probably the success of the low level in the concurrent learning. When learning concurrently with the high level, then the subgoals are being updated to closer/easier subgoals every time a subgoal is not reachable, so the low level is expected to be more successful than in the preliminary acquisition. During the concurrent learning, a lot of mini trajectories end with success as the learning progresses. It is less likely to happen preliminary acquisition, having it performed in the beginning of the training, when the epsilon greedy policy is not sufficiently trained, and explores a lot. Higher success rate is a stronger learning signal, meaning it contributes a lot more. However, asymptotically it seems that not learning during preliminary acquisition configuration, still converges to a similar success rate by 50,000 episodes.

As to the parameters of the **preliminary acquisition**: first, the **"round" acquisition** (back and forth) technique has shown to give a higher added value than flattened acquisition technique. It seems to help the lower level to master many of the possible mini trajectories, which allows the algorithm as a whole to start with a higher success rate. Secondly, the **sample size** during the preliminary acquisition seem to have an effect mainly during the first 30,000 episodes of concurrent learning. From that point and on, we converge to a similar success rate of the algorithm. We have seen that the run-time complexity addition of the preliminary acquisition is not very significant, which strengthens the approach of using it when we want to accelerate the learning. Otherwise, having it trained enough, the advantage of the preliminary acquisition decays. Finally, the **number of repetition** (back and forth) for each sampled goal seem to have only a relatively small effect. It is reasonable to assume that the importance if it is higher for the cases where the number of samples in the preliminary acquisition is relatively small. Having that said, a small choice like 1 repetition results with lowest added value (the learning curve starts relatively low), which makes sense, since we only have one mini trajectory (low horizon primitive actions). A reasonable repetition number would be 5.

In the Minigrid FourRooms environment with preliminary acquisition, the low level success rate showed a curious behavior of a **decrease first and an increase** later-on until reaching the maximum success rate. Trying to reason that, a possible explanation would be that at first the algorithm suggests relatively easy goals that are in the vicinity, probably in the same room. It is more likely that the vicinity learned during preliminary acquisition is round around each goal, and contains relatively easy paths, since they are more probable to be visited, hence learned too. Let us recall that the initialization of the value function approximators is random. At the moment that we have a successful subgoal suggestion (achieved), the value approximation for this subgoal is significantly better than the rest of the states. It means that the high level suggestions will tend to offer close subgoals, because they will be suggested first (more probable to be reached randomly, hence probable to be suggested). Reaching with exploration to the other room is less probable, which is why it takes more time and exploration during concurrent learning to have those states as a probable suggestion. In the beginning of the concurrent learning we see the high success rate of the low level, having the low level mastering the tasks within the same room. At some point with exploration trying to achieve goals that are further away with the concurrent learning, new states that are close to the pass (between the rooms) and beyond it, are revealed and suggested, and then the learning is again "difficult" to the low level. Difficult in the sense that those are tasks that were not yet mastered by the lower level. That is why we see a decrease, having the low level failing to reach the more difficult goals. Then we see an increase again, having the low level mastering the inter-rooms tasks.

In other aspects of the algorithm analysis there are also some important points to notice. The **search space for hyper-parameters is very large**, which makes the process of optimization very hard. Adding the parameters of the preliminary acquisition, with the rest of parameters such as low horizon, high horizon, learning rate, network architecture, we end up trying to optimise in an overly expanded multi-dimensional space. Applying the algorithm in other environments will require this heavy experimentation. However, the algorithm demonstrates that in the long run with sufficient episodes, the preliminary acquisition parameters do not result in big differences. That means that the **algorithm is robust** to the parameters selection for the preliminary acquisition. The parameters that seem to have the major effect are the **horizons** of both low policy and high policy.

The fact that the low horizon of 4 has shown the best performance proves that learning small tasks is helpful, expressing the exact purpose of the **hierarchical learning**. It shows that the algorithm **benefits from the hierarchy**.

The `is_reachable` set and the `goal_list` are key players in the algorithm. The "decision" of who is the next one to act - the high level or the low level, depends on the knowledge accumulated in `is_reachable` exclusively. In addition, the `goal_list`, should be diverse enough and to cover as much of the goal state as possible, since all the subgoal suggestions come from there (except the times where the current goal was never visited, and we ask the explicit policy representation for a suggestion). The fact that those two lists are needed for a good performance of the algorithm might harden the generalizability of the algorithm in more complex environments. With that said, thanks to the `goal_list` we **avoid the unfeasible states suggestion** problem, which makes it very important component.

In **Minigrid Empty Room** 15x15, the simpler environment tested, **RCVL** showed similar performance to the State of the Art algorithms of `DDQN` and `DDQN` combined with `HER`. In **Minigrid FourRooms** 15x15, which is the more complex environment (with obstacles) we aimed to solve, **RCVL has shown superiority** over `DDQN`, `DDQN` with `HER` and the baseline `LRGP`. It has shown faster learning, and efficient subgoals proposing converging to a success rate of approximately 0.975. The paths the algorithm chooses are close to optimal, verifying observing it in action. Recall that this thesis aimed to propose an algorithm for solving complex problems, which is why a minor path suboptimality below State of the Art (though same success rate) in the simple environment is acceptable, as long as in complex environments, it stands out with superior performance.

One major drawback is that it will be hard to apply in **continuous environments**. The algorithm was built to solve discrete environments, but it will have a problem to generalise to the continuous case. It will require some kind of quantization of the state/action/goal space, or other metrics for distance evaluation.

To conclude, the new suggested algorithm RCVL was able to outperform State of the Art methods and the baseline LRGP algorithm in the more complex environment. It manages to solve the problem of unfeasible subgoals suggestion, and reaches an incredibly high success rate. In addition, the paths drawn by the high level seem to be close to optimal when observing it in action. Furthermore, It has shown to have robustness to changes in a lot of the parameters, making it more reliable.

6.2.2 Future work

First of all, the algorithm should be **tested on new and more complex environments**, to demonstrate its generalizability, to make sure the high performance is not for the specific problem we solve with the mini grid world.

Secondly, to expand the algorithm to the continuous space. To apply the algorithm in a **continuous environment**, one option is to discretize the state space. That raises the need to come up with a generalizable framework for **environment discretization**, with minimal harm to the problem resolution. It should be a modular addition, such that there is no need to change the environment itself. It seems like a complicated task, and hard to make in a generalizable manner, since it is most probable that each continuous environment will require different adaptations. Another option is to save the continuous states as we do with the discrete environment, but to use some **metric to estimate the proximity** to them when needed. That metric can also be learned/estimated.

A possible suggestion to overcome the limitation of `goal_list` and `is_reachable` is to approximate those by a neural network, specifically **GAN**, having the generator generating states in the vicinity of a goal, while the discriminator learning to differentiate if a suggested state is in the vicinity or not and feasible or not. The collected experience can be divided into factuals and counterfactuals to train the discriminator. However to create the pool for factuals will be a lot easier than to create the counterfactuals pool in the aspect of reachability. We can only collect states that were visited, but we can't collect those that are not feasible. We can try to collect the cases where the low level performs a forward step but stays in the same location. Then some kind of generalizable estimation of which is the **state** that we are trying to get but can't, should be done, so we can use this state as a counterfactual for feasibility.

To make the process even **more curricular**, instead of suggesting the lowest cost subgoal, it can be a state within thresholds (Q_{min}, Q_{max}), such that the suggested subgoals are always in intermediate level of difficulty, since they have not been mastered yet, however still result with more than a minimal value. That is more similar to the original suggestion of the Reverse Curriculum idea. The challenge in that case would be to set the thresholds for intermediate proficiency level of the lower level.

Bibliography

[1] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[2] Marcin Andrychowicz et al. "Hindsight experience replay". In: *arXiv preprint arXiv:1707.01495* (2017).

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Mario Martin. *Lecture notes in Reinforcement Learning Course*. Mar. 2020. URL: `https://www.cs.upc.edu/~mmartin/ATCI-RL.html`.

[5] Shweta Bhatt. *Reinforcement Learning 101 - Learn the essentials of Reinforcement Learning!* URL: `https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292`. (accessed: 13.05.2022).

[6] Ajitesh Kumar. *Reinforcement Learning Real-world examples*. URL: `https://vitalflux.com/reinforcement-learning-real-world-examples/`. (accessed: 12.05.2022).

[7] 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`.

[8] RAM SAGAR. *What Is Model-Free Reinforcement Learning?* URL: `https://analyticsindiamag.com/what-is-model-free-reinforcement-learning/#:~:text=%5C%E2%5C%80%5C%9CModel%5C%2Dbased%5C%20methods%5C%20rely%5C%20on,methods%5C%20primarily%5C%20rely%5C%20on%5C%20learning.%5C%E2%5C%80%5C%9D&text=In%5C%20the%5C%20context%5C%20of%5C%20reinforcement,be%5C%20made%5C%20about%5C%20the%5C%20environment.`. (accessed: 13.05.2022).

[9] OpenAI. *Spinning Up in Deep RL!* URL: `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html`. (accessed: 24.05.2022).

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[11] Chris Yoon. *Towards Data Science: Double Deep Q Networks*. URL: `https://towardsdatascience.com/double-deep-q-networks-905dd8325412`. (accessed: 02.06.2022).

[12] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[13] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[14] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[15] OpenAI. *OpenAI Spinning Up: Soft Actor Critic*. URL: `https://spinningup.openai.com/en/latest/algorithms/sac.html`. (accessed: 04.06.2022).

[16] Andrew Levy, Robert Platt, and Kate Saenko. "Hierarchical actor-critic". In: *arXiv preprint arXiv:1712.00948* 12 (2017).

[17] Andrew Levy et al. "Learning multi-level hierarchies with hindsight". In: *arXiv preprint arXiv:1712.00948* (2017).

[18] Ofir Nachum et al. "Data-Efficient Hierarchical Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: `https://proceedings.neurips.cc/paper/2018/file/e6384711491713d29bc63fc5eeb5ba4f-Paper.pdf`.

[19] Vivienne Huiling Wang et al. "Adversarially Guided Subgoal Generation for Hierarchical Reinforcement Learning". In: *arXiv preprint arXiv:2201.09635* (2022).

[20] Tianren Zhang et al. *Generating adjacency-constrained subgoals in hierarchical reinforcement learning*. Tech. rep. 2020. arXiv: `2006.11485`. URL: `https://github.com/trzhang0116/HRAC.`.

[21] Tianren Zhang et al. "Adjacency constraint for efficient hierarchical reinforcement learning". In: (2021). arXiv: `2111.00213`. URL: `http://arxiv.org/abs/2111.00213`.

[22] Junsu Kim, Younggyo Seo, and Jinwoo Shin. "Landmark-Guided Subgoal Generation in Hierarchical Reinforcement Learning". In: NeurIPS (2021), pp. 1–14. arXiv: `2110.13625`. URL: `http://arxiv.org/abs/2110.13625`.

[23] Yoshua Bengio et al. "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[24] Jeffrey L. Elman. "Learning and development in neural networks: the importance of starting small". In: *Cognition* 48.1 (1993), pp. 71–99. ISSN: 0010-0277. DOI: `https://doi.org/10.1016/0010-0277(93)90058-4`. URL: `https://www.sciencedirect.com/science/article/pii/0010027793900584`.

[25] Tambet Matiisen et al. "Teacher–student curriculum learning". In: *IEEE transactions on neural networks and learning systems* 31.9 (2019), pp. 3732–3740.

[26] Bhairav Mehta et al. "Curriculum in gradient-based meta-reinforcement learning". In: *arXiv preprint arXiv:2002.07956* (2020).

[27] David Held et al. "Automatic goal generation for reinforcement learning agents". In: (2018).

[28] Carlos Florensa et al. "Reverse Curriculum Generation for Reinforcement Learning". In: CoRL (2017), pp. 1–14. arXiv: `1707.05300`. URL: `http://arxiv.org/abs/1707.05300`.

[29] Carlos Florensa. *Reverse Curriculum Generation for Reinforcement Learning Agents.* URL: `https://bair.berkeley.edu/blog/2017/12/20/reverse-curriculum/`. (accessed: 20.05.2022).

[30] Rafel Palliser Sans. *Learning recursive goal proposal: a hierarchical reinforcement learning approach.* URL: `https://upcommons.upc.edu/handle/2117/348422`. (accessed: 10.06.2022).

[31] Rafel Palliser Sans. *gym-simple-minigrid.* URL: `https://github.com/rafelps/gym-simple-minigrid`. (accessed: 06.06.2022).

[32] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym.* `https://github.com/maximecb/gym-minigrid`. 2018.

Appendix A

Implementation Details

`is_reachable` is stored as a set of tuples with the form of (s, g) where each of them has a 3 dimensional descriptor of (x,y,direction). In both proposed algorithms `RCRL` and `RCVL` we use the same structure of it.

A.1 RCRL implementation details

RCRL was implemented with the default parameters as appear in the table A.1.

Hyperparameter	**Value**
High policy and Value estimation	SAC A.4
Low policy	ϵ greedy DDQN A.3
Low horizon	4
High horizon	15
Learning rate	3e-4
Batch size	256

Table A.1: RCRL default parameters used in studies and experiments

A.2 RCVL implementation details

RCVL was implemented with the default parameters as appear in the table A.2. In the preliminary studies some of the parameters were changes according the described in the specific study, while the rest of the parameters stay the same.

Hyperparameter	**Value**
High policy and Value estimation	SAC (A.4)
Low policy	ϵ greedy DDQN (A.3)
Low horizon	4
High horizon	15
Learning rate	3e-4
Sample size in preliminary acquisition	0
Batch size	256
Radius for neighborhood	1
Symmetry neighborhood collection	False
Flatt or round acquisition (Pre.Acq)	Round
Back and forth repetitions (Pre.Acq)	5

Table A.2: RCVL default parameters used in studies and experiments

The implementation of `goal_list` includes a list in the size of $m \times n$, having m board width and n board height. A function converts from location as (x,y) to a unidimensional indices list. In the case of the board with size of 15x15 we get a `goal_list` with the length of 225. Each ”element” in this list is a set. That assures that there are no repetitions. This set contains the collected states in the vicinity of a goal, with their 3 dimensional descriptors: (x,y,direction).

A.3 DDQN implementation

DDQN was implemented according to [1] with the following hyper parameters:

Hyperparameter	**Value**
Hidden dimension of NN	(128, 128, 128, 128)
Optimizer	AdamW (default)
ϵ	Decay 0.65 to 0.1
Learning rate	3e-4
Buffer size	5e5
Discount coefficient γ	1
Soft-update parameter τ	0.005

Table A.3: DDQN default parameters used in studies and experiments

A.4 SAC implementation details

SAC was implemented based on [14]. It was used for both `RCLV` and **RCRL**, while for the first uses it's `Q-value` approximators, and the actor only for the case where the goal was never visited before so there are no possible suggestions in `goal_list`. The hyperparameters for it are the following:

Hyperparameter	**Value**
Hidden dimension of NN	(128, 128, 128, 128)
Optimizer	AdamW (default)
Learning rate	3e-4
Buffer size	1e6
Discount coefficient γ	1
Soft-update parameter τ	0.005
Entropy regularizer α	1

Table A.4: SAC default parameters used in studies and experiments