Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Improving sample efficiency with Hierarchical Reinforcement Learning in ride-sharing applications

Fernando Victoria

May 2, 2021

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
BAI (Computer Engineering)

Declaration

I hereby declare that this Final Year Project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Ride-sharing applications such as Uber and Lyft have increased in popularity in recent years. These companies employ different types of data-intensive procedures to match drivers with passengers and dynamically price each trip. With the vast volume of data available now a days, Machine Learning techniques are being used for the purpose of optimizing the algorithms used by these companies.

This project examines the application of Reinforcement Learning (RL) in a ride-sharing environment. This setup requires immediate decision making for drivers to pick-up and serve the user's request. On this basis, a RL technique was chosen, as it is concerned with how agents make decisions in a situation. More specifically, a Hierarchical Reinforcement Learning (HRL) algorithm is examined and extended to adapt to the ride-sharing situation. The aim is to speed-up the training process of the algorithm. As HRL divides a problem into a series of sub-problems, if the combination of these can be represented more compactly than the whole problem, this results in a decrease of computational complexity.

The algorithm decomposes the tasks in a 3-level hierarchy. Thus, the initial problem is divided into a set of three sub-problems. The purpose of this is to solve the sub-problems individually and combine them in order to improve the sample-efficiency.

The algorithm is evaluated in different ride-sharing scenarios and compared to a non-hierarchical version. For each experiment, the average and standard deviation of the scores achieved during the training process are examined. This is done to assess how quickly the algorithms learns as well as how reliable they are.

Acknowledgements

I would like to thank my supervisor Ivana Dusparic for providing guidance and support. She was always available and provided me with any help I needed. Moreover, she made sure that I understood the full extent of the project and what was expected of me.

This is also for my family and friends, especially Stefano and Kris, who I want to thank for being with me for 4 years, and making this last year special!

Contents

List of Figures

1 Introduction

1.1 Motivation

The way we travel around cities has been revolutionized in the past years with the rise of ride-sharing services such as Lyft and Uber. The latter reported in 2019 that they operated in 69 countries, had a 68% increase in the number of consumers with respect to the precedent year, and a revenue of $14.1 billion. (Uber (5)).The key factors for building these successful companies have been several. Examples of these include the integration of Google Maps into their app to see the vehicles available, the comfort of automatic online payment, and the optimized algorithms for dynamic pricing and travel routes.

Machine Learning (ML) is at the core of these algorithms, which have benefitted with the growth of Big Data in order to enhance their forecasting and decision-making systems, such as vehicle-passenger matching. In fact, ML is becoming more and more popular in our daily lives, ranging from the research community to businesses using it as their main qualities. This area of Computing, which deals with algorithms that develop automatically by using data with the minimum human intervention, can be categorized as supervised or unsupervised learning. Recently, a different form of ML has been earning popularity, as it thought to revolutionaries the way we build autonomous systems (Arulkumaran et al. (6)): Reinforcement Learning (RL). This field is concerned with how intelligent systems make decisions in an environment and maximize the rewards that it receives for taking actions.

Reinforcement Learning has had recent breakthroughs in recent years, such as or defeating the world champion Go player (Silver et al. (7)) or achieving superhuman performance in multiple classic Atari games (Mnih et al. (8)). Hierarchical Reinforcement Learning (HRL) is a branch of RL which solves a RL setting by decomposing it into a hierarchy of subtasks, and has shown significant success at solving complex problems (Nachum et al. (9)). The main perk of HRL is the substantial reduction in computational complexity when representing the problem as a combination of sub-problems, thus, minimizing the time that the agent spends learning an optimal behaviour.

One of the main application fields of HRL has been solving navigation problems. Examples of this are teaching an agent reach a destination by moving blocking objects along the way, or teaching an agent gather different objects while avoiding others (Nachum et al. (10)). The navigation problems in which HRL has implied a speed-up in the learning process over other RL methods has been where the agent must perform extra tasks to find the best possible route.

1.2 Aims and Objectives

The objectives of this project are several:

- Apply a Hierarchical Reinforcement Learning approach to a ride-sharing problem.
- Expand this approach to allow the HRL agent to find the most efficient route to any goal displayed in the environment.
- Modify the original algorithm in order to make the agent first navigate to a subgoal displayed in any state in the environment and then attain the end-goal in any of the possible locations in the Grid-World.
- Show that by applying a HRL approach, the agent can converge to an optimal policy using fewer episodes (requiring less data samples to converge) than a non-hierarchical approach.

Overall, the aim of the project is to start from the Hierarchical Q-Learning (HierQ) algorithm (Levy et al. (4)) and implement it in a ride-sharing environment. The reason this algorithm was chosen is because it learns the multiple policies in parallel and it substantially speeds-up the learning process with respect to other hierarchical and non-hierarchical algorithms. In Chapter 4, the HierQ baseline is presented, as well as the extensions and modifications applied to it in order to fulfil these objectives.

1.3 Thesis contribution

The most important contribution of this project is the experimental proof that the Hierarchical Q-Learning approach presented is able to improve the sample efficiency in comparison to the flat agent (Q-Learning) in a ride-sharing environment. While Reinforcement Learning has been applied previously to ride-sharing applications, it is to the best of our knowledge, the

first attempt to improve the speed of learning. HierQ was initially trained to reach one fixed goal by solving sub-goals in that same direction. However, it was not trained to solve for any goal in the environment (varying states from one episode to another), nor to solve for sub-goals that were in the goals direction. Therefore, we take the agent as if it was a vehicle, the sub-goal as if it was a passenger, and the final goal as the destination in which the passenger must attain (3-level hierarchy). Finally, the results are compared to the flat agent in different Grid-World scenarios in which there is a vehicle, a passenger and a destination.

1.4 Report Overview

Chapter 2: Introduces the fundamental background information in the fields of Machine Learning, with particular focus on Reinforcement Learning, Hierarchical RL and the environments in which these algorithms are tested.

Chapter 3: Presents state of the art techniques that are relevant to the project, such as Actor-Critic methods, recent research in HRL and the Hindsight Experience Replay technique.

Chapter 4: Describes the design of the algorithm, by presenting the original one and the extensions and adaptations to the proposed environment.

Chapter 5: Discusses the objectives of the experiments, the tools used to carry out these experiments and the experiments themselves.

Chapter 6: Closes the report with a conclusion and possible future work that can be built on top of this project.

2 Background

2.1 Machine Learning

Machine Learning is the field of Computer Science in which algorithms are studied with the aim of providing computers the ability to efficiently do some tasks without explicitly programming the machine to do so. ML algorithms use training and validation data in order to build a model that will predict or make decisions about future outcomes of a real-world problem. Machine Learning can be split into three main fields: supervised, unsupervised and reinforcement learning. In supervised learning, the input data is a 'labelled' dataset containing input-output pairs. With the aid of an optimization function, the loss function iteratively learns to reduce the error in the predicted output and updates the model in order to maximise the accuracy of the prediction. In unsupervised learning, the goal is to find out what can be learned from an 'unlabelled' dataset. Unlike the previous type of learning, this technique performs particularly well in clustering unknown data, K-means being one of the most popular algorithms for this. The third type of ML, Reinforcement Learning, will be introduced more in-depth in Section 2.3.

2.2 Deep Learning

Deep Learning is a specific category of Machine Learning based on Artificial Neural Networks; computing structures inspired by the biological neurons in the cerebral cortex. Each neuron typically receives a weighted sum of inputs and is then passed through a non-linear function, called activation function, with the purpose of determining what to be fired to the next neuron. Neural Networks (NN) gained popularity over the recent years with the rise of Big Data. In comparison with other Machine Learning algorithms, Neural Network's performance scales effectively with the amount of data. Deep Learning has made key advances in the fields of Machine Translation (Bergen et al. (11)), Bioinformatics (Wallach et al. (12)) and Computer Vision (Krizhevsky et al. (13)), among other areas.

2.3 Reinforcement Learning

2.3.1 Introduction

Reinforcement Learning is the third Machine Learning paradigm, together with Supervised and Unsupervised Learning, and it is concerned with sequential decision making. It consists in an *agent* changing from state to state at each time step by executing actions (decided by a *policy*) in a given *environment*.

- *Agent*: The system controlling the main object (for example, a robot).
- *Environment*: The world in which the agent will be. It simulates an ecosystem for the agent in which all objects that the agent can interact with belong to the environment. This component of RL is needed to test the performance of an agent in a particular situation.
- *Policy*: The algorithm that the agent will follow in order to decide which actions to perform.

For every action taken by the agent in a specific state, it receives a positive or negative scalar reward signal, indicating how well the agent is doing at that step. RL is like trial and error learning, the agent's aim is to maximize the total reward by exploring or exploiting the actions in the environment, with the goal of finding the optimal agents behaviour function in the environment: the policy.

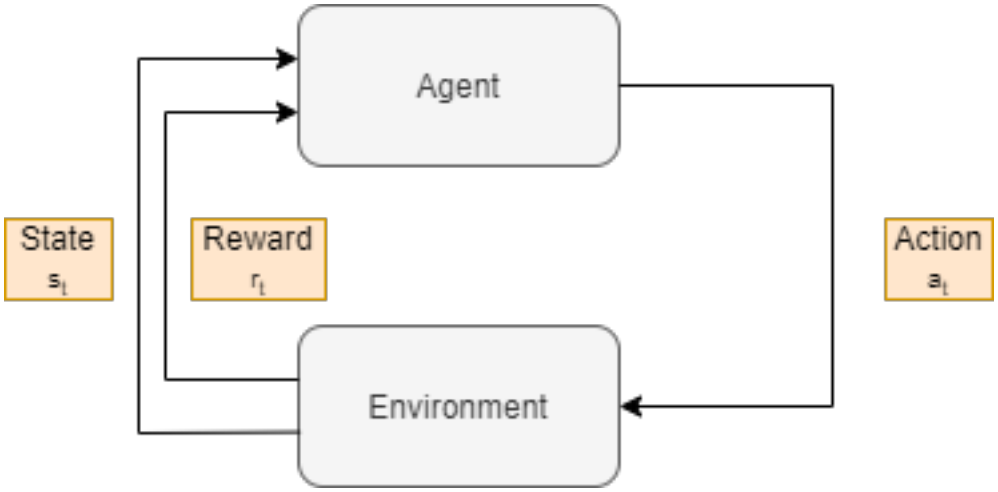

Figure 2.1: Interaction between Agent and Environment in a typical Markov Decision Process scenario.

In order to assess how good or bad a state is, the value function $V(s)$ is used. This function is an estimate of the cumulative reward that the agent can expect by being in a state and acting according to a policy π through all sequential states.

$$V(s) = E(\sum_{t=0} \gamma^t R(s_t, a_t) | s_0 = s) \qquad \forall s \in S \tag{1}$$

Where γ represents a discount factor which will lower the rewards that are further in time and is in the range $[0, 1]$. $R(s, a)$ is the reward function which outputs a scalar value when the agent performs an action a to the environment, and transitions from state s to s'. The action a_t is chosen following the policy π at s_t: $a_t = \pi(s_t)$.

To evaluate how good is it to take a particular action in a particular state, the action-value function $Q(s, a)$ is used. For a given policy π, state s and action a, it computes the expected return from that point onwards, following the given policy. The difference between $Q(s, a)$ and $V(s, a)$ is that the first one allows you to pick an initial action which might be different from the one indicated by the policy.

$$Q(s, a) = E(\sum_{t=0} \gamma^t R(s_t, a_t) | s_t = s, a_t = a) \qquad \forall s \in S \tag{2}$$

The *Figure* 2.2 shows the visual representation of the relationships that produce the update operations. These updates transmit information back to a state or a state-action pair from the following states:

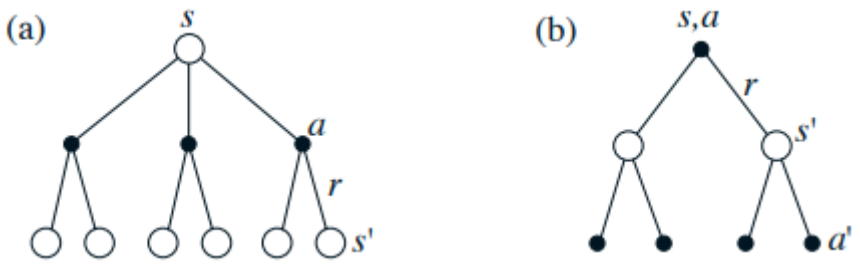

Figure 2.2: Backup diagrams for (a) v_π and (b) q_π (Sutton and Barto (1)).

2.3.2 Exploration vs Exploitation

Reinforcement Learning in many cases learns online, which means that the agent gathers data and learns from it at run time, which presents an important issue when making decisions.

This is whether the agent should exploit the current policy and receive the immediate reward, or take a random action and explore the environment, thus, seeking to gather sufficient information to make the best global decisions.

In order to evaluate how good an online policy is, we must find a suitable performance metric. The *regret* is the difference between the total reward of the optimal policy π^* and the total reward that gathered π.

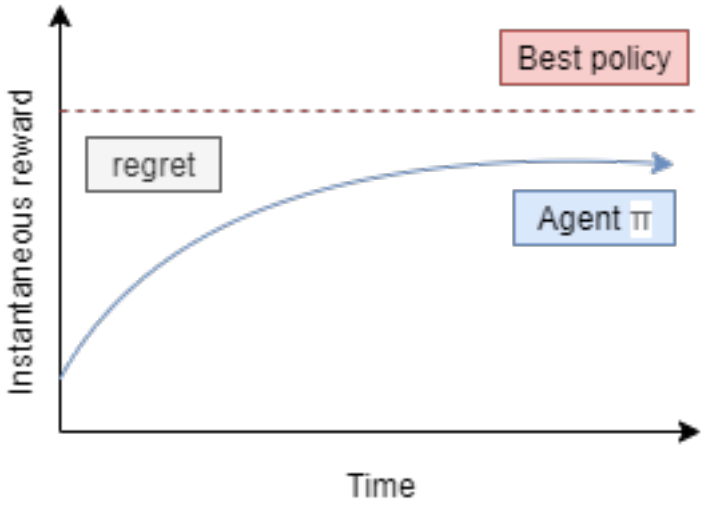

Figure 2.3: Graph indicating the regret between the policy π and the best policy π^*. The regret is the area between the best policy and the agent π.

There exist several solutions to this dilemma, in which a policy achieves a healthy combination (balance) between exploration and exploitation. An example is the *Decaying Epsilon Greedy Policy*, in which the agent explores (takes random action) the environment with a probability of ϵ and exploits the policy π with a probability of $(1 - \epsilon)$. ϵ starts at a high value in the range of $[0, 1]$, and at each iteration, it is linearly reduced. This change in the value of ϵ ensures that the agent dedicates more time to exploring the environment at the beginning, and with time it will converge to an optimal policy.

2.3.3 Markov Decision Processes (MDPs) and Universal MDPs (UMDPs)

Markov Decision Processes are a mathematically idealized form of the RL problem for which precise theoretical statements can be made (Sutton and Barto (1)). MDPs are tuples $(S, A, P_{s,a,s'}, R, \gamma)$ where:

- S is a finite set of States.
- A is a finite set of Actions.
- $P_{s,a,s'}$ is the state transition probability matrix. It maps the state, action pairs to a probability distribution over the set of states.
- R is the reward function.
- γ is the discount factor.

Initially, we start in a state $S_0 \in S$ and choose an action $a_0 \in A$. Consequently, we transition to the state $S_1 \in S$, and the cycle restarts.

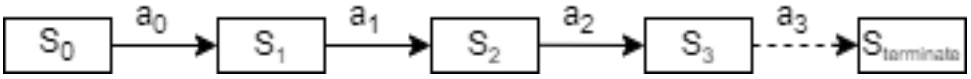

Figure 2.4: Sequence of states and actions.

The goal of Reinforcement Learning is to maximise the total rewards. The total Reward for the sequence of states and actions is given below:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1} \tag{3}$$

For any Markov's Decision Process...

1. There is always a deterministic optimal policy π^* that gives better or equal reward to all the other policies, $\pi^* \geq \pi, \forall \pi$
2. All π^* accomplish the optimal state-value function, $V_{\pi^*}(s) = V^*(\text{s})$
3. All π^* accomplish the optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$

The optimal policy can be defined as:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if a} = \ \text{argmax}_{a \in A}\, q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

Universal Markov Decision Processes (UMDPs) are MDPs amplified with a set of goals $\mathcal{G}$ (each a state or a set of states) that we want the agent to learn. A UMDP can be defined by the following tuple U = (S, G, A, $P_{s,a}$, R, γ), in which a goal $g \in G$ is defined at the beginning of each episode (this goal will last for the whole episode). In order to solve a UMDP, a control

policy $\pi : S, G \rightarrow A$ such that it maximises the following value function for an initial state and goal pair:

$$v_\pi(s, g) = \sum_{n=0}^{+\infty} \gamma^n R_{t+n+1} | s_t = s, g_t = g \tag{4}$$

2.3.4 Model-Based vs Model-Free

Generally, there exists two types of Reinforcement Learning: Model-Based and Model-Free. The first one is the ideal scenario for a RL problem to be solved, as the state transition probability matrix P_{sa} is provided and the agent does not have to interact with the environment during the learning phase. This is advantageous because the agent does not need to wait for the environment to react to its actions, however, if the model is not accurate, we have the risk of learning something that does not correspond to the real environment.

As opposed to Model-Based, Model-Free Reinforcement Learning is not provided with the state transition probability matrix P_{sa} that defines the problem to be solved (Markov Decision Process). Most RL problems are of this type, in which the agent must interact with the environment (trial-and-error) to learn a policy without the help of a model that guides it choose the best actions to perform in each state. While executing actions in the environment during the learning phase, the agent gathers rewards (positive or negative), and updates its value functions. To keep track of these experiences, the agent can build a 'cache' in which it will store the best actions to be taken corresponding to each state. This information is based on the greatest rewards received while interacting with the environment. The disadvantage of this method is that the 'cache' to be stored escalates proportionally to the states and actions of the environment. This is particularly bad for the cases in which the environment in which the agent interacts is large and a significant number of actions must be stored for all states.

2.3.5 Dynamic Programming

Dynamic Programming (DP) makes reference to a set of algorithms that can be used to solve model-based Reinforcement Learning problems. In other words: the structure of the MDP must be known. DP breaks down the problem into smaller sub-problems, and then combines the solutions to the sub-problems to obtain the global solution. This method is used for planning in a MDP, which means that the agent performs computations with its model, without any external interaction, to find the optimal policy (hopefully). DP can address two major classes of problems in a Markov Decision Process:

1. *Prediction*: Given a Policy π and a MDP, evaluate the value function V_π in order to assess how good that policy is. To do this, iterative methods using the Bellman equation can be used, such as the Iterative Policy Evaluation:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow ... \rightarrow v_\pi \tag{5}$$

The initial approximation (v_0) can be randomly initiated. For example, having all values set to 0, indicating no rewards. In order to obtain the consecutive approximations v_1, v_2, etc, the Bellman equation is used:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')], \qquad \forall s \in S \tag{6}$$

When this sequence approximates to $k \rightarrow \infty$, it can be shown that it converges to v_π, and this is called *Iterative Policy Evaluation*.

2. *Control*: Given a MDP, find the optimal policy π_* that will maximise the cumulative rewards. To do this, the method described above for policy evaluation is used, and in order to improve the policy π, we will have to act greedily with respect to the value function of π iteratively until the policy π converges to π_*, the optimal policy. In this context, acting greedily means determining the best action which will result in the maximization the action value function:

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}}\, q_\pi(s, a) \tag{7}$$

In Reinforcement Learning, the prediction problem is typically solved to solve the control problem.

2.3.6 Monte Carlo

Monte Carlo (*MC*) learning methods are useful for estimating value functions and finding optimal policies when the environments dynamics are not completely known. They require a model, however, unlike Dynamic Programming, the model only needs to generate sample transitions that will compute through experience, in other words, it is sample-based learning. Recall that in DP, the structure (transition probability matrix) of the MDP had to be known.

In Reinforcement Learning, experience is defined as "sample sequences of states, actions and rewards from actual or simulated interaction with an environment" (Sutton and Barto (1)). In order to ensure well-defined returns, we define Monte Carlo approaches exclusively for episodic tasks, which means that we assume experience is separated into episodes, and that all episodes conclude at one point, regardless of the actions. It is when an episode terminates that these methods will re-compute their value and policy estimates, thus, MC only works for terminating environments. This is called off-line learning, when the computation occurs in an episode-by-episode manner. The other type of learning is called on-line learning, and refers to methods that learn from partial returns, in a step-by-step behaviour.

The idea behind all Monte Carlo methods is that the state-value function $V(s)$ can be learned by averaging the returns observed after visits to a particular state. As the number of returns increase, the average should converge to the expected value. One method to compute $V(s)$ is called the First-Visit MC method, which estimates $v_\pi(s)$ as the average of the returns after first visits to s:

Algorithm 1 First-visit MC prediction, for estimating $V \approx v_\pi$

Input a policy π to be evaluated
Initialize $V(s) \in \mathbb{R}$, arbitrarily, for all s $\in S$
Initialize $Returns(s) \leftarrow$ an empty list, for all s $\in S$
Loop forever (for each episode):
    Generate an episode following π: S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T
    G $\leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, ..., 0$:
        G $\leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, ..., S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$

2.3.7 Temporal-Difference Learning

Temporal-Difference (TD) methods learn directly from episodes of experience, like Monte Carlo, the do not require prior knowledge of the environment dynamics (model-free). The difference between TD Learning and MC is that the first one learns on-line from incomplete episodes, called bootstrapping, as opposed to MC, which only learns when the episode terminates. When evaluating the policy, the simplest TD technique makes the update instantly when transitioning to S_{t+1} and receiving R_{t+1}:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1} - V(S_t)] \tag{8}$$

This method is called one-step TD, or TD(0). It is a special case of TD(λ), $\lambda \in [0, 1]$, where depending on the value of it, it indicates how shallow or deep the algorithm bootstraps. In the case that $\lambda = 1$, it is in fact the Monte Carlo algorithm, as it indicates that the algorithm will update the value-function when all steps terminate, ie: after the episode terminates.

In order to have a better understanding of the difference between Monte Carlo and Temporal Difference Learning, consider the following scenario: You are driving a bicycle, there is a car coming straight at you at a high speed and you are going to have an accident. At the last second, the driver turns, and the car avoids you. In Monte Carlo, you would not update the reward as a negative one (nearly had an accident). You would wait till other car passes, and when one of them hits you, have an accident, and the episode terminates (you stop cycling), then it would update the reward as a negative one.On the other hand, TD(0) would update as soon as the car avoids you. When it has passed, it would receive a negative reward and it would update the value-function, resulting in a change of optimal strategy which could be to slow down the bicycle speed.

As previously mentioned, bootstrapping occurs when the update involves an estimate, and not a fully accurate return. It is one of the dimensions that have to do with the type of update employed to improve the value function, that is, to the depth of updates. Dynamic Programming and Temporal Difference use this technique to evaluate the value function. On the other hand, Monte Carlo methods require to go to the end of the episode to have the update, so they do not bootstrap. As it can be seen in Figure 2.5, the degree in which TD(λ) bootstraps depends on the value of lambda (depth, vertical axis).

Sampling is another dimension that describes the type of update used to improve the value function. As seen in Figure 2.5, the horizontal axis indicates whether the algorithms use sample updates (sample paths in the environment), or full backups (based on distribution of possible trajectories). Monte Carlo and TD methods do sample, as they do not require to do a full width exhaustive search. On the other hand, Dynamic Programming does not sample.

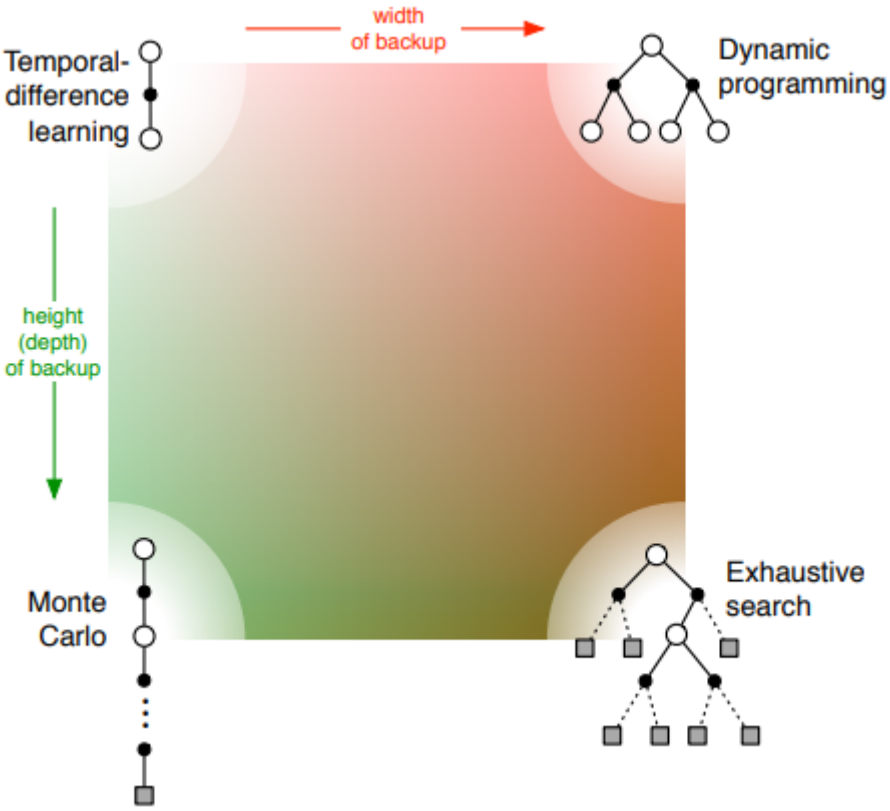

Figure 2.5: A unified view of the space of Reinforcement Learning methods.

2.3.8 Q-Learning

Q-Learning is a model-free Reinforcement Learning algorithm that seeks to find the optimal policy π_* that will indicate the best possible actions in each state. The agent computes the instantaneous reward at each state, which will depend exclusively on the action and the state. It learns off-policy, which means that the policy that the algorithm is evaluating is not the same as the one describing the behaviour of the agent. In other words, following a policy $\mu(a|s)$, evaluate policy π. Some of the advantages of off-policy learning are that the agent can re-use experience generated from old policies, it can learn about the optimal policy while following an exploratory policy and learn about several policies while following another one.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{9}$$

In Q-Learning, the agent explores all states in numerous occasions, and employs different actions in each individual state. The corresponding values for each state-action pair that

result from Equation 9 are saved in a $|S|x|A|$ matrix that acts as a cache and only stores the best actions to take in each state.

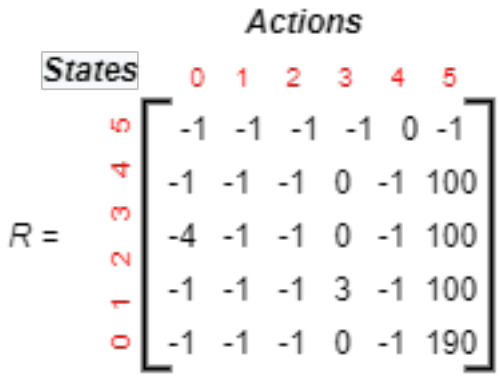

Figure 2.6: An example of a $|S|x|A|$ Q-table containing 6 states and 6 actions.

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning is the combination of Deep Learning and Reinforcement Learning. When the environment of the Reinforcement Learning setup is too large (high dimensionality), this technique lacks scalability and cannot solve the problem efficiently. Deep Reinforcement Learning overcomes this problem thanks to The Universal Approximation Theorem (Hornik (14)) which says that "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units". In other words, the fundamental property of Neural Networks is that they can find compact low-dimensional features in high-dimensional input data, such as images, audio, videos. However, it has been showed in recent works (Szegedy et al. (15), Mhaskar and Poggio (16)) that deeper networks generalise better and need fewer parameters.

In the context of Reinforcement Learning, instead of having to store a lookup table with the mapping of the states to values, or state-action pairs to Q-values, Neural Networks learn to approximate the state-value function V(s), action-value function Q(s, a), policy function, etc. In order to train the Deep Neural Networks, gradient-free optimizers and gradient-based optimizers have both been successful for policy search methods (Deisenroth* et al. (17)), however, gradient-based methods have been the preferred approach for Deep Reinforcement Learning, as it is more efficient when the policy contains large number of parameters.

2.5 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) tackles Reinforcement Learning (RL) problems by decomposing them into sub-problems. In this category of RL, the agent divides a task into a hierarchy of sub-tasks, such that the high-level sub-task invokes lower-level sub-tasks, forming a hierarchy of smaller problems to solve. HRL deals with the problem of dimensionality that RL suffers: "the number of parameters to be learned grows exponentially with the size of any compact encoding of a state" (Barto and Mahadevan (18)).

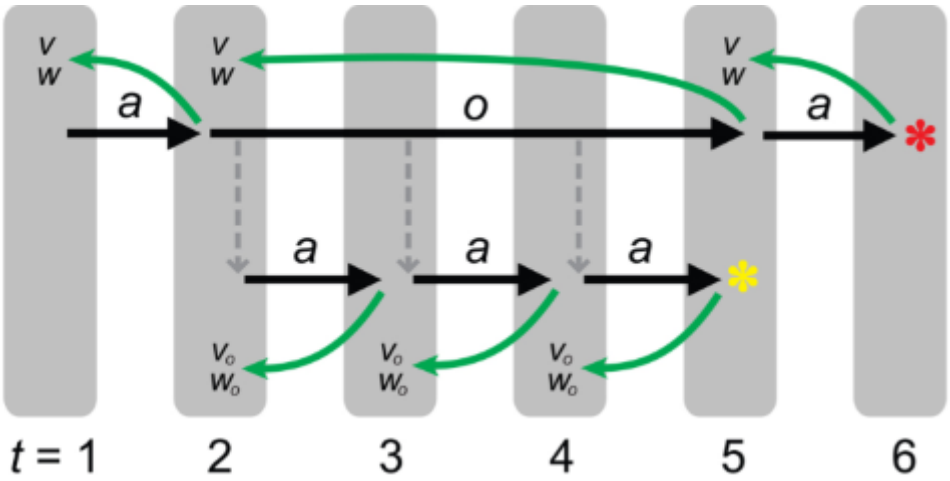

Figure 2.7: A schematic representation of Hierarchical Reinforcement Learning. At $t = 1$ the agent executes an action, and based on the following state (at $t = 2$), a feedback error (green arrow from *state* 2 to *state* 1) is computed in order to update V and W (the action/option weight). The long black arrow going from $t = 2$ to $t = 5$ is an option. During these 3 time-steps, the actions (three short black arrows) are chosen from the option's policy. The green arrows are the errors computed after each action is executed and they update the action weight (W_0) and option-specific value (V_0). When the option's sub-goal is reached, the options terminates and an error is computed (long green arrow), which will update V and W from the state it initiated (in this case $t = 2$).The asterisks are the rewards that will be useful for assessing if an action or sequence of actions was good at a specific level (Botvinick et al. (2))

When complex problems can be decomposed in smaller problems in a hierarchical manner, the time and space complexity of the initial challenge might be reduced. This can lead to an improvement in the learning and performance of the whole task, as multiple levels of policies can be trained simultaneously.

For example, imagine a setting in which a person is in a couch in the living room of a house and is hungry. The goal of the person would be to eat something so the hunger goes away. Instead of remembering an exact set of actions to satisfy the person's hunger, he/she will break down the problem into different sub problems and solve those smaller sub-tasks in order to solve the initial task. In this setting, the person's goal is clear: to eat something.

A sub-task that the person would need to solve would be to go to the kitchen, where the food is located. As he/she is in the living room, a previous sub-task that would need to be solved would be to get out of that room, and then go to the kitchen. This hierarchy of tasks, starting from the most distant one, breaking it down to smaller sub-tasks, would derive in the person having to take an action in the environment, in this case the real-world. An example of the lowest-level policy that would output an action in the environment would be to move the muscles of the body. By doing this in a particular order, the person would be able to use the policies learned at each level of the hierarchy, so the movement of the muscles solve the problems of: standing up, going out of the room, going to the kitchen, and finally, eating.

2.6 Reinforcement Learning testing environments

2.6.1 OpenAI Gym environments

OpenAI is a research company with the purpose of supporting and expanding Artificial Intelligence boundaries. Recently, they announced GPT-3 (Brown et al. (19)), an "auto regressive language model with 175 billion parameters, 10x more than any previous non-sparse language model". This model achieves ground-breaking results in Natural Language Processing fields such as question-answering and translation. In 2016, for the purpose of Reinforcement Learning research, they released OpenAI Gym (Brockman et al. (20)).

OpenAI Gym is a Python toolkit that offers different environments on which researchers can deploy their RL algorithms and compare them. The settings on which to implement the algorithms range from continuous control tasks, such as making a 3D two-legged robot walk or making a car reach the top of a hill, to Atari 2600 games, such as Pong or Pinball.

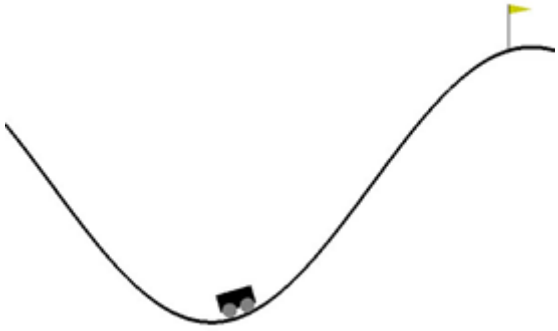

Figure 2.8: *MountainCar* – *v*0 environment in OpenAI Gym. This environment consists in a car positioned between two hills in which the goal is to drive to the yellow flag (right hill). In order to accomplish this, the car must build momentum by going back and forth (the engine is not powerful enough to only drive up the mountain).

In order to evaluate the deployed algorithms, it is essential to get feedback from the actions applied to the environment. For this reason, OpenAI Gym includes the step function. This returns several values when the agent performs an action to the environment, which are:

- *Observation*: An object that is particular to each environment that represent the state of the environment. Examples of this are the state of the board in a chess game or the state of the taxi, passenger and the other essential components of the road.
- *Reward*: The quantity of reward (scalar) that was achieved by performing the previous action. These values vary depending on the environment.
- *Done*: Boolean variable that indicates if the episode terminated (True) or is still going on (False). When the variable is True, the environment resets.
- *Info*: Dictionary variable that contains information about the state of the environment. It is important to note that this is useful for debugging purposes, but the agent is not permitted to use it for learning.

2.6.2 Ride-sharing environments

One of the essential challenges when applying RL to real-world scenarios is the learning environment in which to test the algorithm. As opposed to supervised learning, where the data is static and can be assessed by dividing it into two sets (training and testing), RL presents an interactive method of learning, which adds difficulty when training and evaluating algorithms. While environments for training Machine Learning models for Self-Driving cars have been popular in recent years (Dosovitskiy et al. (21), Kiran et al. (22)), there is a lack of environments for Reinforcement Learning in the area of ride-sharing. In this section, several environments used in RL ride-sharing research will be presented to give a better understanding of the settings in which RL algorithms have been deployed.

The main environment in which RL algorithms have been tested for ride-sharing applications has been grid-worlds (Vasebi and Hayeri (3), Singh et al. (23)) and network models made up of nodes and links to represent the location (Maciejewski and Nagel (24)). For the gird-world type of environment, the location (a city, for example) is divided into squares (Figure 2.8) which represent the states in which the vehicles can be. This compression of the representation of the city is done to facilitate the training task.

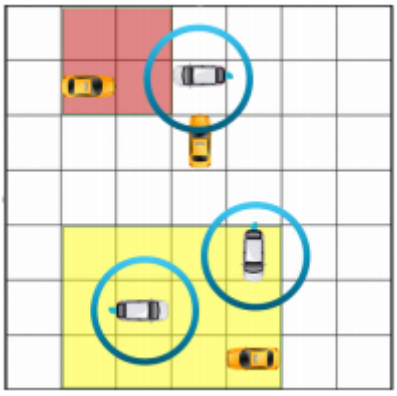

Figure 2.9: Example of grid-world environment in a ride-sharing application (Vasebi and Hayeri (3))

Another aspect to consider when evaluating ride-sharing algorithms are the dynamics of the environments. The orders of passengers and the trajectories of vehicles are an important factor, as they will differ heavily from one city to another. In order to train and evaluate the Reinforcement Learning algorithms, real-world traffic and passengers demand can be integrated in the environment in order to calibrate it. This can be done using historical data from companies such as Didi Chuxing (Qin et al. (25)).

3 State of the Art

3.1 Actor Critic methods

Actor-Critic methods are a subcategory of Temporal Difference methods (Section 2.3.7) that represent independently the policy and the value function with the help of a distinct memory composition. It is composed of two main parts:

- **Actor:** The policy structure, which performs actions to the environment.
- **Critic:** Estimated value function that assesses how good or bad are the actions performed by the Actor.

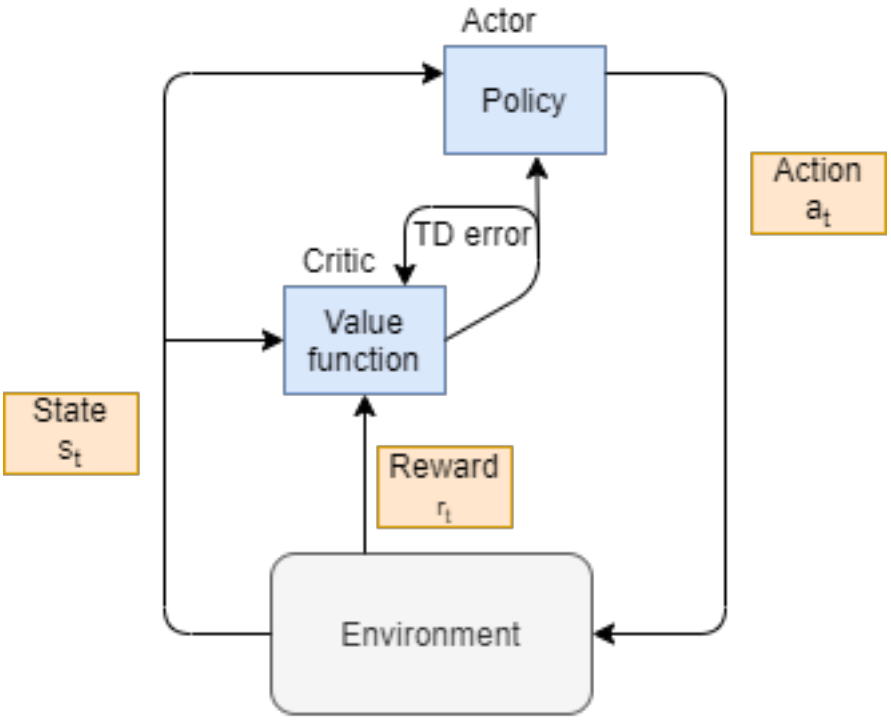

Figure 3.1: Actor-Critic diagram.

In this type of Reinforcement Learning, the actor follows a policy, and the critic must learn and critique the policy with the help of TD error (1), thus, being an on-line learning method. The Critic evaluates how good the action taken by the Actor was and determines if the new state is better or worse than the past state.

$$\delta_t = r_{t+1} + \gamma V(s_{t+1} - V(s_t) \tag{1}$$

Where δ_t is the Temporal Difference (TD) error which measures the difference between the estimated value of S_t and the better approximation $r_{t+1} + \gamma\ V(s_{t+1})$ (Sutton and Barto (1)).One thing to note about this error measurement is that it depends on next reward (r_{t+1}) and next step (s_{t+1}), which indicates that the error is not available until one step later in the episode.

3.2 State of the art in Hierarchical Reinforcement Learning

Developing HRL agents has been an enduring topic in the field of Reinforcement Learning (Precup (26), Dayan and Hinton (27), Dietterich (28), Boutilier et al. (29), Vezhnevets et al. (30)). Indeed, exploring frameworks that combine action-value functions that operate at distinct temporal dimensions has been the study topic for researchers in companies like Google (Nachum et al. (10)), Facebook (Sukhbaatar et al. (31)) and Microsoft (Liao et al. (32)), which have had applications in disease diagnosis (Liao et al. (32)), video captioning (Wang et al. (33)) and navigation tasks (Li et al. (34)). A popular formulation for Hierarchical Reinforcement Learning has been the options framework (Precup (26), Sutton et al. (35)), which introduces a termination policy for each option (sub-policy) in the hierarchy. The options framework depends on of previous knowledge for creating options, however, in (Bacon et al. (36)) an Actor-Critic algorithm is proposed in order to simultaneously learn them with the higher-level policy. In end-to-end Hierarchical Reinforcement Learning, (Bacon et al. (36)) has meant great progress. Nevertheless, the Option-Critic approaches tend to learn either one policy that operates throughout the entire episode, or a sub-policy that terminates every time step. The problem now is not on how to learn options, but on what good options should be learned, and it has been showed in several papers such as (Vezhnevets et al. (30)) and (Harb et al. (37)) that regularizers are crucial on learning useful sub-policies.

Other approaches to Hierarchical Reinforcement Learning were the MAXQ value function decomposition, introduced by Thomas G. Dietterich in (Dietterich (28)) and Hierarchies of

Abstract Machines (HAMs), presented by Ronald Parr and Stuart Russell in (Parr and Russell (38)). Together with the Options framework explained above, these three methods are the building blocks of HRL, presented between 1997 and 2000.

Recently, there has been several other Hierarchical Reinforcement Learning approaches that work in discrete and continuous domains. In (Konidaris and Barto (39)), Skill Chaining was introduced, which consists in a "skill discovery method for continuous domains". This approach consists in a 2-level HRL architecture that chains options from the end goal-state to the start-state incrementally. FeUdal Networks (FUNs) (Vezhnevets et al. (40)) were introduced in 2017 and supposed a novel architecture for HRL. These were inspired by Feudal Reinforcement Learning and the framework consists in a Manager and Worker module in which the Manager sets abstract goals which are given and performed by the Worker, similar to Actor-Critic methods. One of the key advantages of FUNs is that they not only allow long timescale credit assignment, but they also allow different sub-policies to emerge, depending on the goals set by the Manager. Finally, another approach that needs to be mentioned is presented in (Nachum et al. (10)). Here, the off-policy learning algorithm HIRO (Hierarchical Reinforcement Learning with Off-policy correction) presents an architecture where the low-level controllers are regulated with the goals that the higher-level controllers propose and learn. The Off-Policy correction that is incorporated in the model allows the algorithm to learn all the policies at the different levels in fewer interactions with the environment, thus, speeding up the process of learning.

In this project, we use Hierarchical Reinforcement Learning to improve the speed of learning an effective algorithm in ride-sharing applications. While Reinforcement Learning has been applied in ride-sharing applications, this is the first attempt, to the best of our knowledge, to improve the speed of learning in RL-based ride-sharing domain

3.3 Hindsight Experience Replay

Hindsight Experience Replay (HER) (Andrychowicz et al. (41)) is a novel technique that allows sample-efficient learning from sparse and binary rewards. The idea behind this method is to allow the algorithm to learn from failures, as well as from success. For example, imagine a setting in which you are trying to kick a football ball from the penalty point into the goal. If you score, you receive a goal for your team (positive reward), and if you do not score, you receive no reward. If this task was performed by an algorithm and threw the ball to the left, it would only conclude from that episode that the chain of actions that it took, led to a failure (not scoring). By doing this, the algorithm learnt nothing, however, humans

could have concluded from the failed episode that if the goal was moved to the left, the kick would have been a success and it would had scored. To try to accomplish this type of logical thinking, HER uses goals in order to expand the reward signal. The policies and value functions that are trained with HER have as input a state and a goal, as opposed to only having a state. This generalisation over state and goal pairs comes from the Universal Value Function Approximators (UVFAs) (Schaul et al. (42)), which factors observed values into different state and goal vectors, and then learns to map from the state and goal pairs to the factored embedding vectors. It is assumed that there is a function that maps the states of the environment into a binary reward system (1 or 0), and the end-goal of the RL agent is to attain a state that $f_g(state) = 1$. However, the efficiency of this technique is not optimal, as the rewards do not give enough information to the agent (only gets 1 or 0).

To optimize this technique, HER uses a particular method to sample new goals. It starts with the agent adding to the experience buffer the transitions in which the agent acts into the environment (this is common in other similar algorithms). However, the particular method in HER samples different goals (in addition to the ones achieved by the agent) and adds them to the buffer with their corresponding variants every time an episode terminates. There are many approaches on how to compute the additional goals added to the buffer for replay, but in all of them, the agent replays the trajectories with the same goal followed in the previous episode. An example of a method to derive new goals is to obtain it from an arbitrary state in which the agent was located in the past episode, but in a transition sequential to the one we are currently evaluating.

4 Design

4.1 Problem formulation

The problem that we want to tackle is related to ride-sharing. In our problem statement, a vehicle is in charge of picking up different passengers and delivering them according to their destination points. Figure 4.1 presents a setting in which a vehicle must pick up two passengers and deliver them to their desired destinations (presented with a cross and the corresponding number).

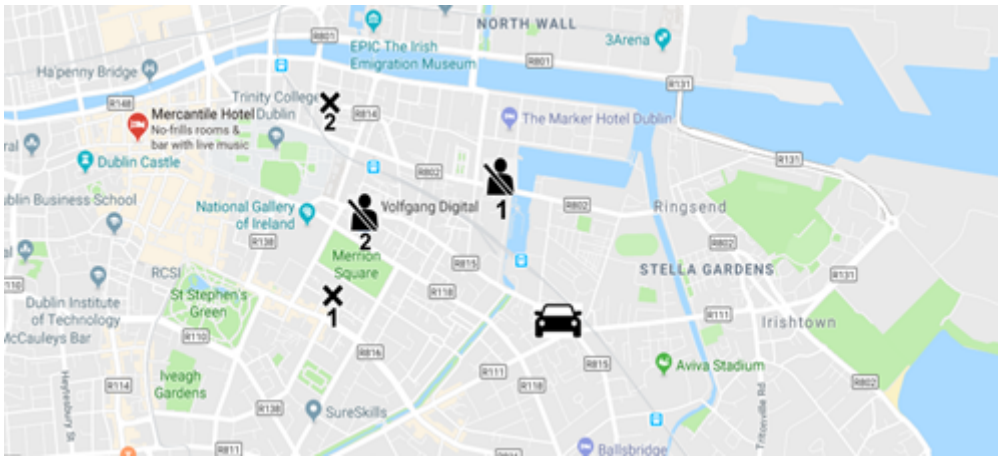

Figure 4.1: Map showing the problem that is being solved: How to pick-up and deliver passengers efficiently in a ride-sharing environment.

The problem for vehicles, or companies like Uber or Lyft, is to optimize the travel time of the vehicles and to efficiently pick-up and deliver both passengers (up to four passengers per car) to their destinations. In this project, we tackle the issue of learning new policies for picking-up and delivering passengers quickly, thus, speeding up the training process of the algorithm.

4.2 Taxi-v3

Section 2.6.1 introduced OpenAI Gym, a toolkit providing different environments to test Reinforcement Learning algorithms on. In Dietterich (28) the "Taxi" task was introduced to present some matters regarding Hierarchical Reinforcement Learning. This discrete environment (Figure 4.2) consists of a 5x5 Grid-World in which a vehicle (indicated by the yellow rectangle) must learn to pick-up and deliver a passenger to a destination.

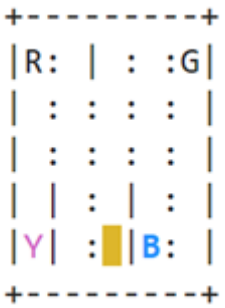

Figure 4.2: *Taxi* − *v*3 environment from the OpenAI Gym.

As seen in the above image, there are four locations specified with a letter: (R, G, Y, B), and the goal of the agent (taxi) is to pick-up a passenger in one of these locations and drop him/her off in another. The taxi has a set of six actions to complete this task: move left, move right, move up, move down, pick-up and drop-off. Each time the episode starts, the taxi initiates in a random rectangle in the environment, and the passenger in one of the four specified locations. The state space of this environment is represented by the row in which the taxi is located as well as the column, the location of the passenger (any of the specific locations, including inside the taxi), and the destination of the passenger. The episode ends once the taxi drops-off the passenger in the correct destination.

For the rewards in the environment, the agent receives a negative reward of -1 each time-step it takes (this is done so the algorithm learns to take the fastest route to complete the task, not just a random one). It will receive +20 points for a successful drop-off and -10 points if the taxi decides to pick-up or drop-off in an "illegal" location (for example, dropping off or picking-up a passenger in none of the four locations indicated by the letters.

4.3 Baseline

This section describes in detail the design of our Hierarchical Reinforcement Learning algorithm. Since this project expands on HierQ algorithm, an adaptation of the HAC (Hierarchical Actor-Critic) (Levy et al. (4)) algorithm but in discrete environments, we will describe in further details how our baseline works. Furthermore, the extensions to the algorithm that have been implemented as well as the modifications applied in order to suit our particular environment will be presented.

4.3.1 HierQ

Our baseline is an off-policy multi-level hierarchy algorithm that can learn efficiently the levels thanks to two main components: a specific structure of the architecture of the layers in the hierarchy and a method for learning the levels of the hierarchy simultaneously and independently (Levy et al. (4)). Hierarchical Q-Learning (HierQ) is presented as a transformation procedure to a UMDP, explained in section 2.3.3.

The HierQ algorithm transforms the original UMDP, $U_{original}$ = (S, G, A, T, R, γ), into a series of h UMDP's U_0, U_1, ..., U_{h-1}. The purpose of this division of the original UMDP is to efficiently learn h-level hierarchy Π_{h-1} with a specific policy each level(π_0, π_1..., π_{h-1}). An important characteristic of the HierQ algorithm (discrete world) and the HAC algorithm (continuous world) is that they are able to learn as many levels in the hierarchy as the user chooses.

HierQ enables each level of the hierarchy to learn its own deterministic policy $\pi_i : S_i, G_i \rightarrow A_i, 0 \leq i \leq k-1$. The state space at every level remains the same, meaning that the state space of the original problem is the same as in all the levels of the hierarchy. In addition to this, the space state and the goal state at each level in the hierarchy is the same ($G_i = S$), as each policy at each level must learn how to solve a shortest path problem with respect to a goal. Furthermore, the upper levels output a sub goal state to the following next lower levels to achieve. For this reason, the action space at all levels (except the lowest layer) is equal to the next-levels goal space: A_i = S, i > 0. Finally, for the lowest level of the hierarchy, the action space is the same to the primitive actions that the agent can apply to the environment, in this case moving to the right, left, up and down.

In order to decompose a problem into small sub problems, HierQ learns hierarchies of nested deterministic policies. This is a key feature in the algorithm, as it allows the agent to learn multiple policies requiring short sequences of actions instead of long sequences of primitive

actions. By embedding the policy at level $i-1$ into the transition function T, HierQ is able to nest policies. Each sub goal level $i > 0$ contains a transition function T_i in which each level i selects a sub goal action a_i which is assigned to be the goal of the following lower-level layer in the hierarchy: $i-1 : g_{i-1} = a_i$. The policy corresponding to that level (π_{i-1}) then has a maximum of H attempts (variable provided by the user) to reach g_{i-1}. The transition function ends and the current state of the agent is returned when one of the following happens: *i*) Policy π_{i-1} performs its H attempts or *ii*) a goal g_n, n $\geq i-1$ is achieved. Due to the nested architecture of the policies, the transition function T_i at level i depends on the full policy hierarchy below that level. Each action that derives from the policy at level $i-1$ depends on the transition function T_{i-1}, which at the same time depends on the policy at level $i-2$, and so on.

At each level, the Q-learning algorithm is used in order to learn each policy. Furthermore, HierQ uses a pessimistic Q-value initialization to avoid the agents to suggest sub goals that are too far away. Thus, the output of the algorithm is a set of k trained Q-tables $Q_0(s, g, a)$,..., $Q_{k-1}(s, g, a)$, one per level in the hierarchy. Pessimistic Q-value initialization, in this case, means that the Q-table at the top-layer of the hierarchy is initialized to - number-of-squares values, as this would be the worst possible goal for an agent to explore an environment, without going backwards to another state that it has already been. On the other hand, the middle layers Q-tables are initialized to a -*HxH* value, which is the maximum number of steps in which the sub goal should be achieved (again, worst case). Finally, the bottom layer Q-table is initialize with -H, which is the maximum number of actions which the lowest level policy can perform before reaching a new state.

Algorithm 2 Hierarchical Q-Learning (HierQ)

Input:

- Key agent parameters: number of levels in hierarchy $k > 1$, maximum subgoal horizon H, learning rate α

Output:

- k trained Q-tables $Q_0(s, g, a), ..., Q_{k-1}(s, g, a)$

Use pessimistic Q-value initialization: $Q_i(s, g, a) \leq -H^{i+1}$
for M episodes **do** ▷ Train for M episodes
  $s_{k-1} \leftarrow S_{init}, g_{k-1} \leftarrow G_{k-1}$ ▷ Sample initial state and task goal
  ▷ Initialize previous state arrays for levels $i, 0 < i < k$
  $Prev\_States_i \leftarrow Array[H^i]$ ▷ Length of level i array is H^i

  while g_{k-1} not achieved **do** ▷ Begin Training
    $a_{k-1} \leftarrow \pi_{k-1_b}(s_{k-1}, g_{k-1})$ ▷ Sample action using ϵ-greedy policy π_{k-1_b}
    $s_{k-1} \leftarrow train - level(k-2, s_{k-1}, a_{k-1})$ ▷ Train next level
  end while
end for

function TRAIN-LEVEL($i :: level, s :: state, g :: goal$)
  $s_i \leftarrow s, g_i \leftarrow g$ ▷ Set current state and goal for level i
  for H attempts or until $g_n, i \leq n < k$ achieved **do**
    $a_i \leftarrow \pi_{i_b}(s_i, g_i)$ ▷ Sample action using ϵ-greedy policy π_{i_b}
    if $i > 0$ **then**
      $s_i^{'} \leftarrow train - level(i-1, s_i, a_i)$ ▷ Train level $i-1$ using subgoal a_i
    else
      Execute primitive action a_0 and observe next state $s_0^{'}$
      ▷ Update $Q_0(s, g, a)$ table for all possible subgoal states
      for each state $s_{goal} \in S$ **do**
        $Q_0(s_0, s_{goal}, a_0) \leftarrow (1-\alpha) \cdot Q_0(s_0, s_{goal}, a_0) + \alpha \cdot [R_0 + \gamma max_a Q_0(s_0', s_{goal}, a_0)]$
      end for
      ▷ Add state s_0 to all previous state arrays
      $Prev\_States_i \leftarrow s_0, 0 < i < k$
      ▷ Update $Q_i(s, g, a), 0 < i < k$, tables
      for each level $i, 0 < i < k$ **do**
        for each state $s \in Prev\_States_i$ **do**
          for each goal $s_{goal} \in S$ **do**
            $Q_i(s, s_{goal}, s_0') \leftarrow (1 - \alpha) \cdot Q_i(s, s_{goal}, s_0') + \alpha \cdot [R_i + \gamma max_a Q_i(s_0', s_{goal}, a)$
          end for
        end for
      end for
    end if
    $s_i \leftarrow s_i^{'}$
  end for
  return $s_i^{'}$ ▷ Output current state
end function

Figure 4.3: Hierarchical Q-Learning algorithm (Levy et al. (4))

4.3.2 Hindsight Action Transition

As mentioned earlier, one of the key components in HRL is to learn the policies in parallel. In HierQ, there are two reasons of non-stationary transition functions that will need to be overcome in order to let the parallel learning to occur. The first one is the updates to lower level policies, which means that whenever the policy at level i changes, the transition functions at levels higher than $i, P_{j|\Pi_{j-1}}, j > i$, are able to change. Exploring lower-level policies is the second cause. As previously stated, HierQ uses deterministic policies, which means that at every level in the hierarchy, it will need to explore with a behaviour policy π_{ib} which is different than the policy that the level is learning: π_i

To overcome these problems that impede the parallel learning of policies, HierQ trains each subgoal policy in which it assumes that P_i uses the best possible lower-level policy hierarchy: Π^*_{i-1}. By making this assumption, $P_{i|\Pi^*_{i-1}}$ is stationary because it does not depend on the changes and explorations of the lower-level policies. This way, it allows the agent to learn a policy at level i while it also learns the policies below that level.

For better understanding on hindsight actions transitions, Figure 4.4 will be explained. In this setting, we see a 2-level agent starting at state s_0 with the goal of reaching the destination (orange circle). Here, the high-level policy π_1 outputs a sub goal state g_0 for the low-level policy π_0 to achieve. After this, the low-level policy π_0 executes H primitive actions (recall that H is a number specified by the user) using the conduct specified by the behaviour policy π_{0b}. As seen in the image, it is unable to reach the goal g_0 and the agent terminates in state s_1. When the H actions are terminated, the first action by π_1 is complete, and a hindsight action transition can be generated.

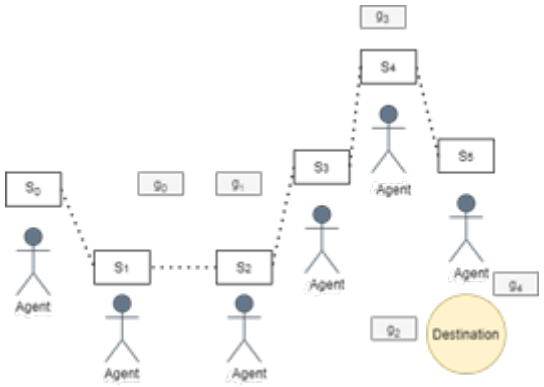

Figure 4.4: Example of the trajectory of a 2-level agent setting.

One of the key elements of hindsight action transitions is that it uses the sub goal state attained in hindsight as the action element in the transition, as opposed of using the original sub goal state that was proposed. For this reason, the hindsight action transition tuple contains the initial state (S_0), action (s_1), reward (yet to be defined), goal (destination circle), discount rate(γ). The reward function used at each sub goal level is the second key component and has two requirements. The first one is that it should incentivize short routes to the goal instead of the long ones. The reason for this is that these shorter paths can be learned faster. The second one is that the reward function must remain independent of the paths carried at lower levels.

Indeed, the objective of hindsight action transitions is to mimic a transition function that uses the best possible policies in the lower levels (Π^*_{i-1}). However, as the dynamics of the environment are unknown, it is unknown what would be the path taken by Π^*_{i-1} and the reward function should only depend on the state reached in hindsight and the state of the goal. If the goal is not achieved at each sub goal level, the agent receives a reward of -1, and in the contrary, a reward of 0 is obtained.

Coming back to the example in Figure 4.4 where the agent tries to reach the destination, the hindsight action transition that the agent would receive would be: [initial state = s0, action = s1, reward = -1, nest state = s1, goal = destination circle, discount rate = gamma]. This tuple would have been the same if the sub goal proposed by the high-level would have been s1 and the optimal lower-level policy hierarchy would have been used to reach it. Following the same procedure, the hindsight action transition tuple generated for the next action by the policy pi_1 would have been [initial state = s1, action = s2, reward = -1, next state = s2, goal = yellow flag, discount rate = γ].

A key point to denote in this method is that the high level of the agent can also learn when the hindsight actions in the episode are -1. The sub goals that are presented by the high-level are discovered through these negative-reward transitions as well, as it will learn to adjust the sub goals to fit in the time scale of the H primitive actions, which is the time size it should be learning. These transitions assume that the lower-level policy is optimal π^*_0 instead of the actual low level policy π_0, thus, allowing more exploration in the lower level policy π_0 as well as being more prone to change.

4.3.3 Hindsight Goal Transition

Hindsight Experience Replay (HER) (Andrychowicz et al. (41)) can be extended to the HRL setting. In (Levy et al. (4)), an additional set of transitions is used to supplement the levels

of the hierarchy. This method, called Hindsight Goal Transitions, allows each level to learn more efficiently in tasks with sparse reward.

In Figure 4.4, a setting was presented where a 2-level agent tried to achieve a destination. In this setting, it could be challenging for any level in the hierarchy to obtain sparse reward. Another easy use of hindsight is the Hindsight Goal Transitions, which ensure that following every chain of actions by each level, that level obtains a transition including the sparse reward. In the example in Figure 4.4, policy π_0 executes at most H primitive actions each high-level action and the low level creates two different transitions:

- 1^{st} **transition**: This transition evaluates the primitive actions taken in the environment given the state of the goal. In the example above, the transition tuple that the low-level would receive after a primitive action would be: [initial state = s_0, action = physical movement, reward = -1, next state = one of the dots between two states, goal = g_0, discount rate = γ].
- 2^{nd} **transition**: This transition is the same as the first one, but the reward and the goal state are unknown. The transition is generated after the H actions (at most) are executed and it produces the missing components in the additional transitions that were made.

Finally, hindsight goal transitions ensure that after every sequence of actions, at least one of the transitions will include the sparse reward. For this reason, hindsight goal transitions should considerably help all the levels in the hierarchy learn a policy that is goal-conditioned.

4.4 Extending HierQ

4.4.1 Architecture

The original HierQ algorithm was programmed to learn for 1 goal. Indeed, the architecture of the layers in the hierarchy (Figure 4.6) only allowed the agent to reach the same goal state when testing the environment, and failed when the goal state changed location. In Figure 4.5, a 2-level agent (blue) tries to achieve a goal (yellow) by suggesting sub-goals that the agent has to achieve (purple). If during the training phase of the algorithm the goal selected was the yellow square in Figure 4.5, if the goal changed during the testing phase (located in another square), the agent would not be able to achieve it (unless it randomly went to the state in which the new goal is located).

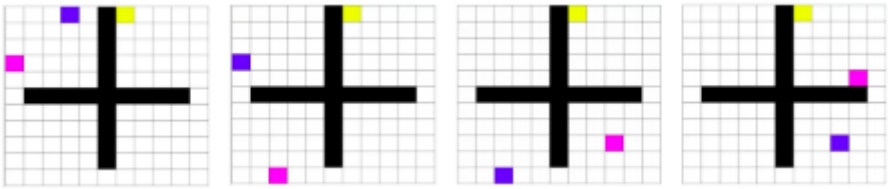

Figure 4.5: Sequence of episodes of a 2-level agent in the environment proposed with HierQ. The blue square is the agent, the purple square is the sub goal, and the yellow square the final goal.

In Figure 4.6 it can be seen that the highest level of the hierarchy, in this case the 3rd level (layer 2), only contains one Q-table which the agent would solve in order to output a sub-goal to the layer above. As previously explained, this table contains the same dimensions as the environment (the number-of-squares x number-of-squares is the same area as the grid-world in which the agent is evaluated). Thus, the individual table only allows the agent to learn values for one goal.

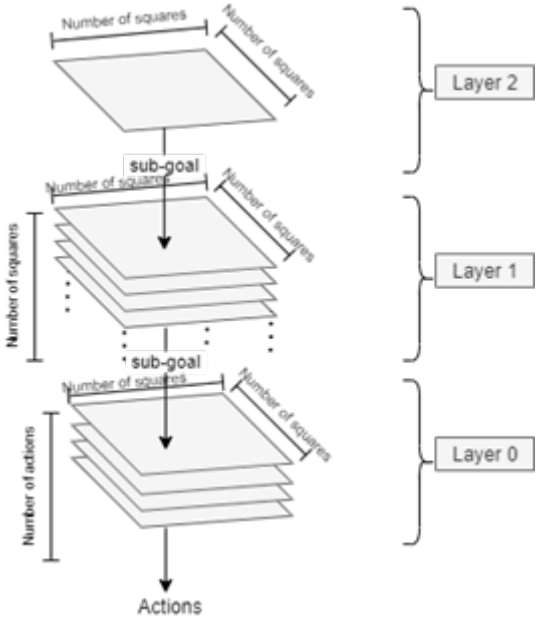

Figure 4.6: Diagram of the architecture of the original HierQ algorithm.

In order to extend the task so the agent could learn for any goal prompted in the grid-world, the architecture had to be changed (Figure 4.7 shows updated architecture). In this case, the top layer was changed so it contained one Q-table per possible goal in the environment

(number of squares possible goals). Now, depending on the goal which the agent must achieve, it access the corresponding table. Then, the layer 2 will output a sub goal ($subgoal_2$) to the layer 1, which will have to solve and output a sub goal ($subgoal_1$), which the layer 0 will have to solve.

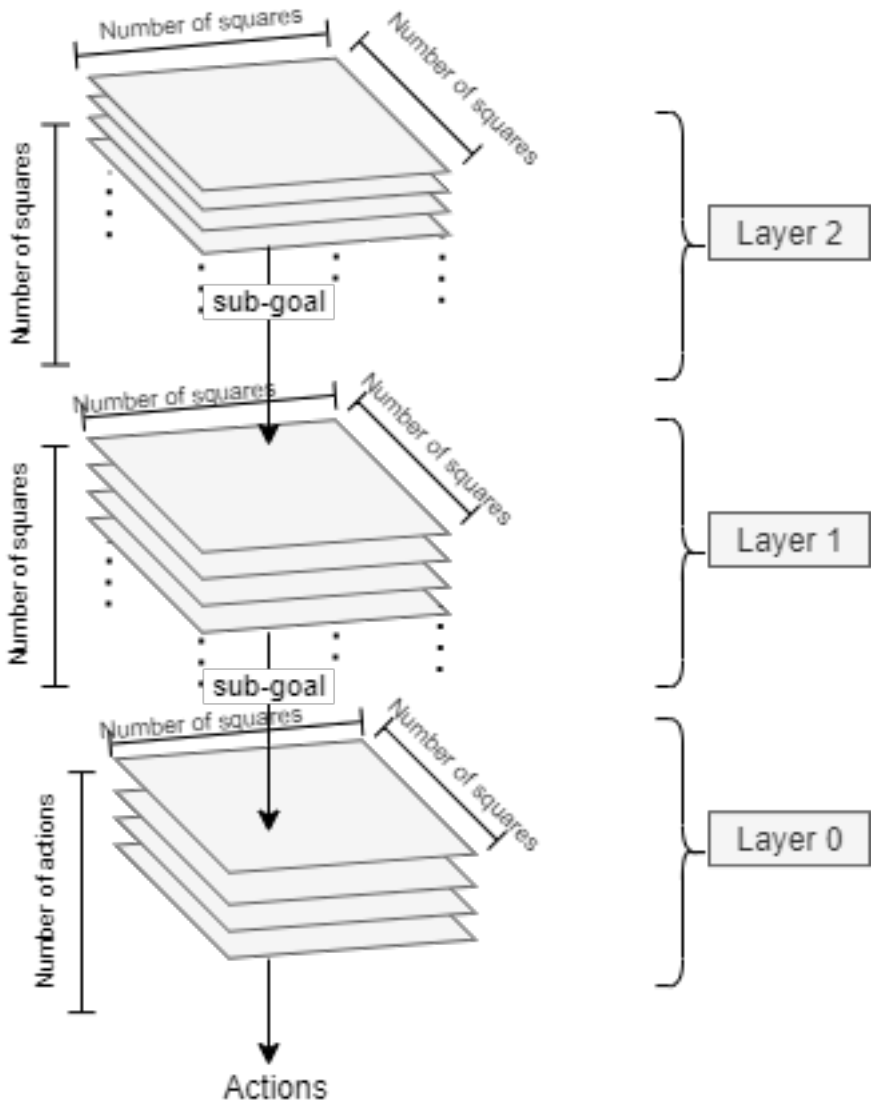

Figure 4.7: Diagram of the architecture of our algorithm.

4.4.2 Environments

Environment 1.

The first environment (Figure 4.8) consists of a 5*x*8 Grid-World similar to the OpenAI Gym *Taxi* − *v*3 environment (Figure 4.2). There are several walls (black squares) that the agent must learn to avoid (and which are not accessible for the agent to be in), and the white squares in which the agent must find the most efficient path to pick-up and deliver the passenger(s). The states are denoted by a number, in which the first row in the environment are states 0-7, in the next row states 8-15, and so on...

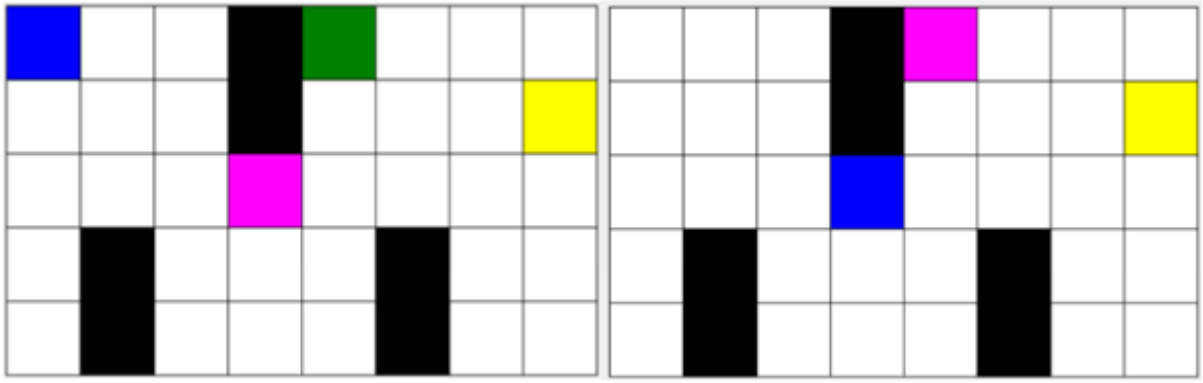

Figure 4.8: Sequence of the first environment used for evaluation.

The blue square indicates the position of the agent (starting position at the top left corner of the Grid). The yellow square indicates the location in which the passenger must be dropped. Note that if the agent (taxi) has not picked-up the passenger and delivers it to the desired destination, no award will be given. The green square reveals the $subgoal_2$, and the magenta square the $subgoal_1$, which will be the input (in addition to the state of the agent) in order to output an action that the agent will take.

Environment 2.

The second environment (Figure 4.9) consists of a 10*x*16 Grid-World (double the size of the previous environment) and contains different wall shapes and sizes. The colours presented in the environment are the same as in the previous one. This environment was selected in order to test the agent in a bigger Grid-World and analyse how a larger environment affected the training process of the agents.

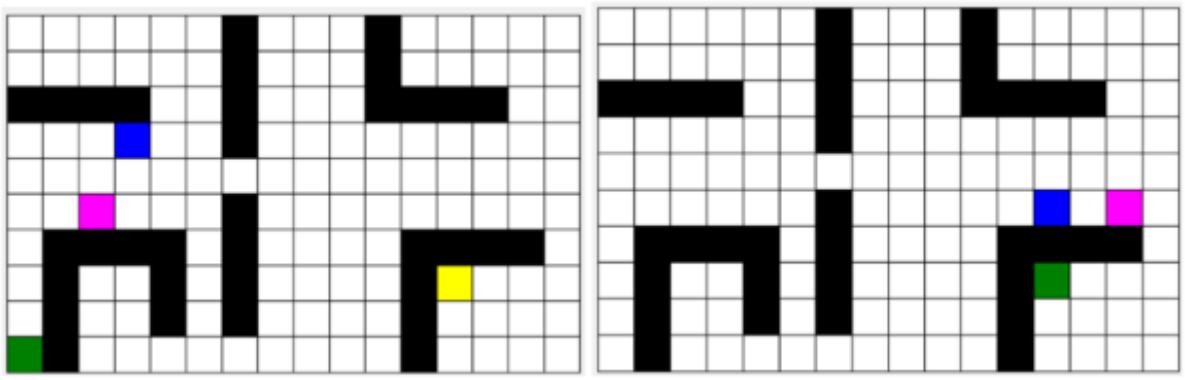

Figure 4.9: Sequence of the second environment used for evaluation.

4.4.3 Other functionalities

Adapting algorithm to ride-sharing environment.

The original HierQ algorithm consists in a variety of layers that all contribute to output sub goals and learn to go in the direction of the goal. Furthermore, the sub goals that the agent suggests contribute to making the agent choose the shortest path.

As the algorithm had to be extended to the ridesharing environment, where a passenger is not necessarily between the between the vehicle (agent) and the desired destination of the passenger, some changes needed to be implemented so the agent could learn to first go in the direction of the passenger and once it was picked, go in the direction of the destination:

- The $subgoal_2$ which was the sub goal the layer 2 (highest level) had to solve first was fixed. As this sub goal represented the passenger, the algorithm would have to work out the path from its state to the state of the $subgoal_2$ initially, in order to pick him/her up.
- Once this sub-task was completed, the algorithm would re-compute the sub goals, and suggests sub goals in the direction of the desired destination of the passenger. By only taking into account the final destination in the second part of the algorithm (once the passenger was picked), we allowed the agent to learn an optimal path for both cases, without interfering one with the other.

Passenger generator

The initial algorithm trained for one goal that was the same for all episodes, specified before the training process. As in our version the goal (destination of the passenger) and passenger location ($subgoal_2$2 of the algorithm) changed from episode to episode in order to learn to solve for every possible combination of passenger/destination in the environment, a passenger/destination generator had to be implemented. This tuple generator is called at the beginning of each episode, and a tuple containing the state of the passenger and the state of the desired destination is generated, depending on the environment selected.

5 Evaluation

In this chapter, the objectives and results obtained will be presented. In order to assess the performance of the algorithm, several evaluations in two different environments have been done and compared against the flat agent (1-level hierarchy), which uses simple Q-Learning.

5.1 Objectives

The main objective of the experiments carried out in this project is to demonstrate empirically that the 3-level Hierarchical Reinforcement Learning agent learns faster than the 1-level simple Q-Learning agent in a ride-sharing environment. We evaluate these two algorithms in the environments presented in Section 4.4.2 and compare them.

Furthermore, the secondary result of this evaluation is to compare the stability of both algorithms. The standard deviation is computed for both algorithms and is also compared, thus, giving us a better insight on the reliability of the algorithms.

5.2 Libraries and settings

The experiments are carried out using the Python programming language in Anaconda, a distribution of Python (among other languages) that seeks to make easier managing packages. For this, an Anaconda environment Python version 3.6 (named "*Py*36") which contained all libraries was created, and it included:

- **Pandas**: An open-source, effective and flexible Data-Frame for data manipulation and dealing with data sets in general in Python. This library is used in this project for storing the values to be plotted (later explained) in a Pandas Data-Frame.

- **CSV**:The CSV (Comma Separated Values) library eases the reading and writing from/to CSV files using Python. It is used to store the Pandas Data-Frames previously mentioned into Excel-generated csv files for storing the values to be plotted in the local computer, thus, being able to reuse the computed values from the runs.
- **Matplotlib**: Open-source visualization library that allows users to create static, dynamic or interactive plots for Python. In our case, this library is used to plot the stored values of the Data-Frames (taken from the Excel file) in order to generated the graphs for the experimental results of this project.
- **Pathlib**: Library that deals with different Operating Systems (in our case Windows) which represents filesystem paths. We use it to manage the different files of the project, particularly to locate the data from the multiple runs of the algorithms in our system.

Other packages that were used for other purposes explained below in the project are:

- **NumPy**: Tool for scientific programming with Python. It offers efficient N-dimensional array manipulation and other essential mathematical tasks. This package is used in areas such as Quantum, Statistic and Astronomy Computing, which require strong computational power. In this project, this library is in charge of dealing with all the creation and updating of the layers in the hierarchy, as well as storing the values in arrays to be later processed with the Pandas Data-Frame.
- **Argparse**: This library eases the writing of user-friendly command-line interfaces. We use it to select the different settings depending on the users preference, which are:

  -- ***show***: This line will render the environment in case the user wants to see the agent during the learning or testing phase.

  -- ***retrain***: As the name indicates, this line will train the agent (learning phase) in the environment. At the end of the training, the agent will store the tables to use them when an optimal policy is needed (testing).

  -- ***test***:As opposed to the command above, this one will test the agent by using the stored tables that were created during the training process.

  -- ***env1***: This line tells the program to choose the environment 1 as the environment.

  -- ***env2***: This line tells the program to choose the environment 2 as the environment.

  -- ***one***: This line indicates that 1 passenger/destination tuple will be present per episode.

  -- ***two***:This line indicates that 2 passenger/destination tuples will be present per episode. However, this is yet to be implemented and can be used for future work, as this setting is not available.

- **Tinker**: Python library to create Graphical User Interfaces (GUIs). It is used to create the window in which the environment and agent are displayed. It creates an appealing framework in which to show the project.

5.3 Experiments

The experiments performed are separated depending on the environment. For each environment, three different tests are done, in which the training episodes are modified from one test to the other, resulting in 6 different tests. As previously mentioned, this project deals with improving sample-efficiency with the use of a Hierarchical Reinforcement Learning algorithm, which means that we want to see a maximum number of rewards in the agent in the least number of episodes. All experiments carried out consist in 1 agent (vehicle), 1 passenger ($subgoal_2$) and 1 destination per episode (final goal). The rewards plotted were done such that each time-step, the agent would receive a -1 reward, and when the passenger was delivered successfully, a reward of 0.

5.3.1 Experiment 1: Comparing training process of HierQ and flat agent in Environment 1

Experiment 1 consisted in training the 3-level hierarchy and the flat agent in the Environment 1 (Figure 5.1). For each test, both algorithms were run 10 times, so the average reward per episode was captured as well as the standard deviation.

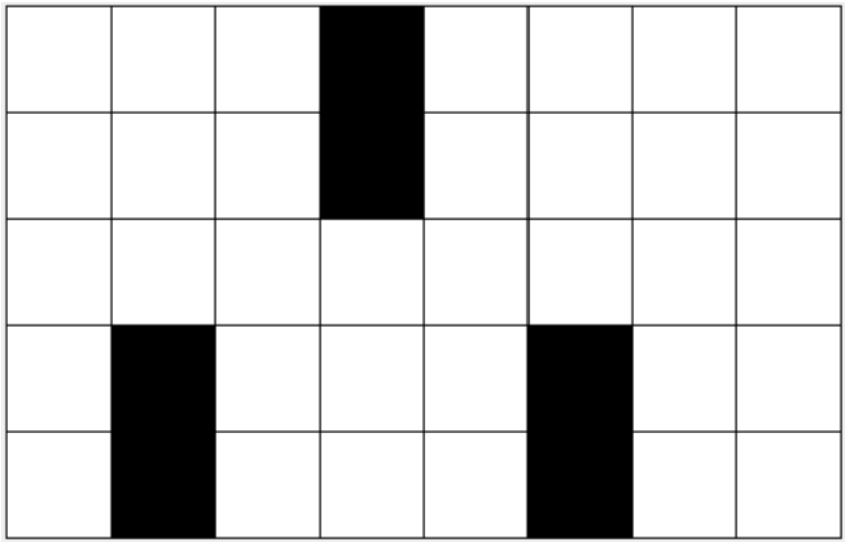

Figure 5.1: Environment 1. White squares indicate where the agent can move and where the passengers and goals/sub-goals can be, and black squares represent invalid states (non-accessible).

Test 1- Environment 1

The first setting in which we train both algorithms is in Environment 1 during 50 episodes (Figure 5.2). Here, we can see that our algorithm (red) is compared to the flat agent (green), a Q-Learning agent. On the x-axis, the number of episodes is showed, and on the y-axis the cumulative reward per episode. As the algorithms uses a pessimistic Q-value initialization, we can see that the flat agent starts at a reward of -40 because the environment has 40 possible states. On the other hand, the 3-level hierarchy agent starts at -70. This is due to the parameter H that we chose for the purpose of the experiment, which indicates the maximum number of actions that the lowest-level policy can perform to reach the sub goal suggested by the one-level-above policy. As this number was set to 5 (maximum of 5 actions), the Q-table was initialized with the value -70, which comes from: -40-(5x5)-5. Furthermore, the standard deviation differs from one algorithm to the other. The hierarchical one presents a lower standard deviation than the flat agent, which varies considerably from episode to episode.

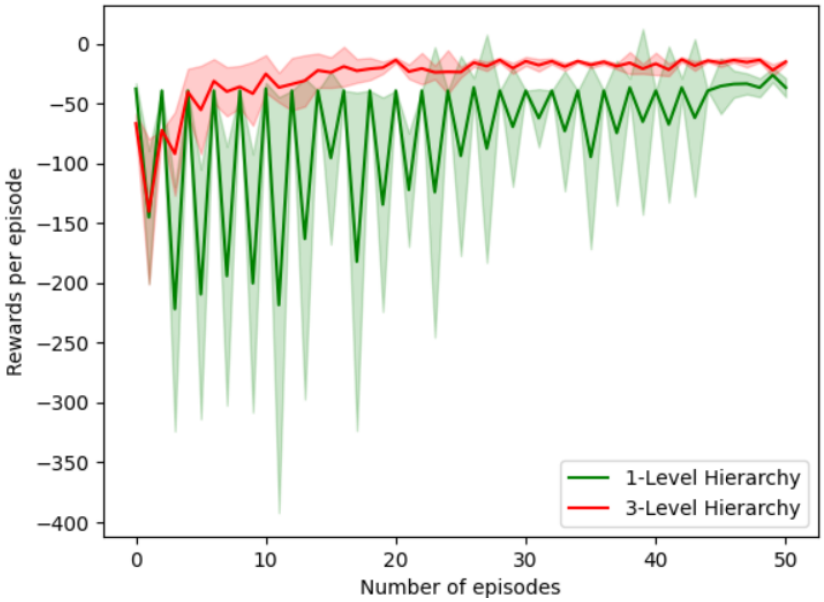

Figure 5.2: First test in Experiment 1: Agents trained in Environment 1 over the course of 50 episodes.

The results obtained show a speed-up in the learning process when using the hierarchical approach with respect to the non-hierarchical approach. Indeed, it is observed that when training both algorithms in the Environment 1 over the course of 50 episodes, the hierarchical approach speeds-up the learning process. Furthermore, it achieves on average more cumulative rewards per episode than the flat agent at all episodes (not only at the beginning). The standard deviation of the algorithms (higher in the flat agent) indicates that the HRL approach is more reliable than the other, as it will score similar rewards when training it several times with the same settings.

Test 2- Environment 1

The second setting consists in training both algorithms in the Environment 1 during 200 episodes. This setting only differs from the Test 1 in the number of episodes.

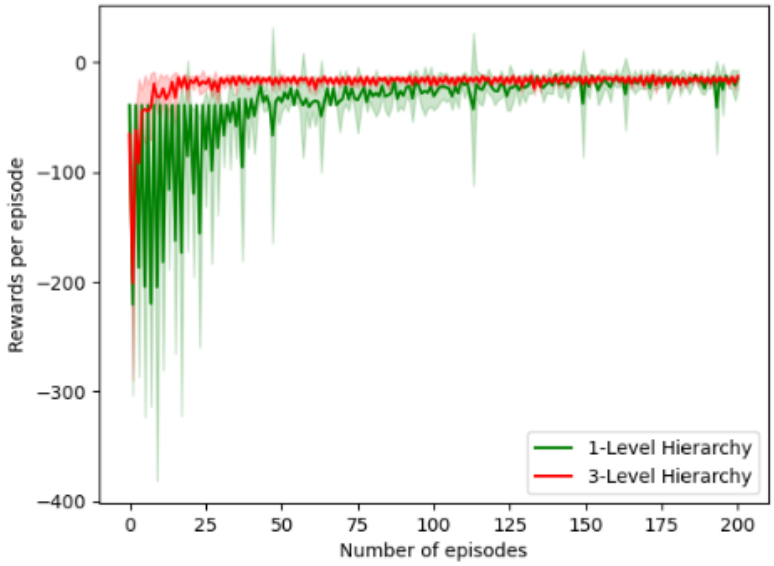

Figure 5.3: Second test in Experiment 1: Agents trained in Environment 1 over the course of 200 episodes.

As for the behaviour of both algorithms, it is showed that the HRL agent indeed learns faster, but with the increase of episodes, both algorithms converge to a similar performing policy. The standard deviation remains high during the first 50 episodes in the flat agent, but then it lowers considerably for the rest of the episodes. By the episodes 175 to 200, both algorithms receive similar cumulative rewards, which imply that the HRL agent does not outperform the flat agent in gathering more rewards, but that it only improves the sample efficiency.

Test 3- Environment 1

The last test performed in the Environment 1 consists in training both algorithms in the Environment 1 during 2000 episodes.

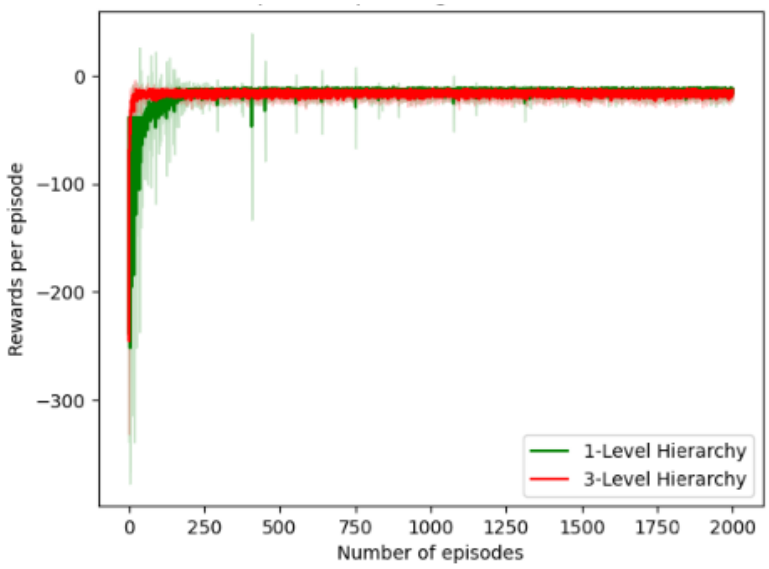

Figure 5.4: Third test in Experiment 1: Agents trained in Environment 1 over the course of 2000 episodes.

The results indicate what could be seen in the previous experiment: the hierarchical approach only speeds-up the learning process but does not achieve a better result than the non-hierarchical approach. In fact, the flat agent learns at a lower rate the best policy to follow, however, it eventually gathers more rewards per episode on average.

5.3.2 Experiment 2: Comparing training process of HierQ and flat agent in Environment 2

Experiment 2 consisted in training the 3-level hierarchy and the flat agent in the Environment 2 (Figure 5.5). For each test, both algorithms were run 10 times, so the average reward per episode was captured as well as the standard deviation.

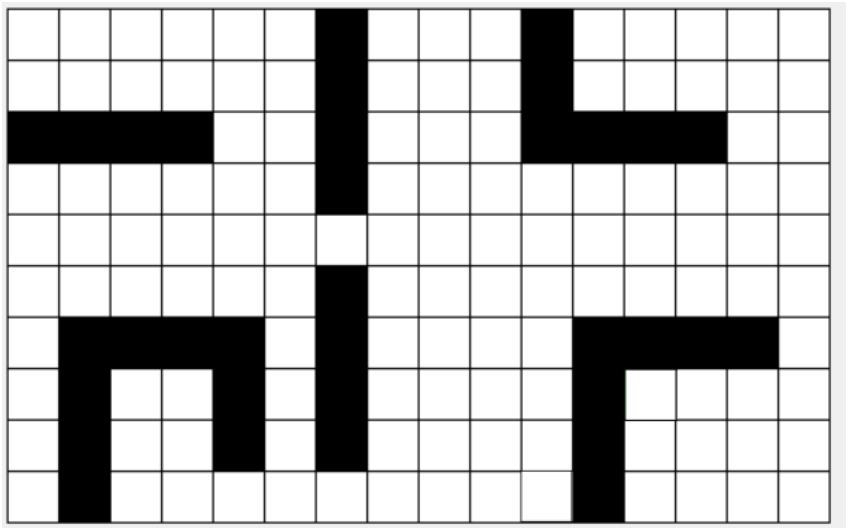

Figure 5.5: Environment 2. White squares indicate where the agent can move and where the passengers and goals/sub-goals can be, and black squares represent invalid states (non-accessible).

The tests carried out in the second environment only differ from the Tests 1-3 already presented in the environment selection. However, this implied a change of the magnitude of rewards. The reason for this is because the states (squares in the two Grid-Worlds) change: the first one is 40, and the second one is 160. Following the same logic as before, with H set to 5, the initial values for the values in the Q-tables for each algorithm were: i)160 for the flat agent and ii)160(5x5)– 5 =190 for the 3-level one.

Test 1- Environment 2

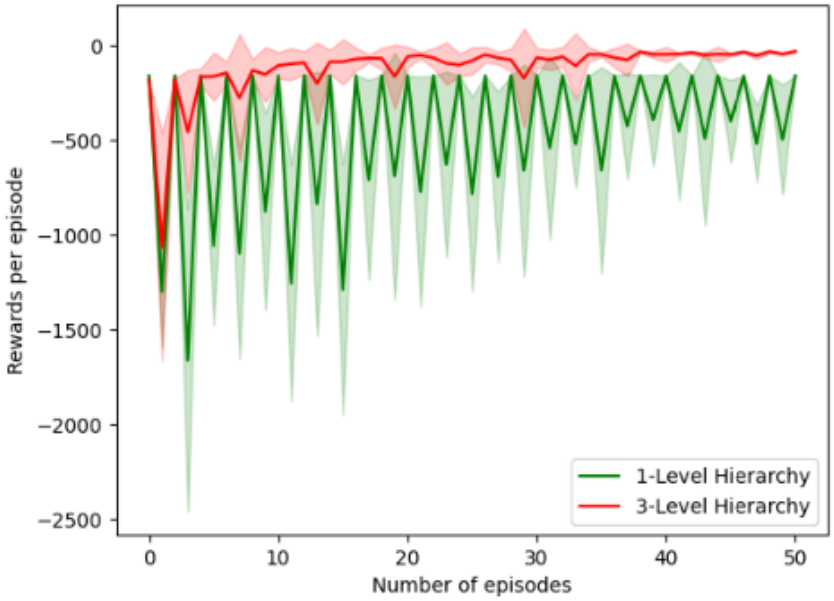

Figure 5.6: First test in Experiment 2: Agents trained in Environment 2 over the course of 50 episodes.

The results show a speed-up in the learning process when using a HRL approach with respect to the flat agent. As it could be seen in the first test of Experiment 1, our method achieves on average more cumulative rewards per episode than the flat agent at all episodes. The standard deviation of the algorithms (higher in the flat agent) indicates that our approach is more reliable than the other, as it will score similar rewards when training it several times with the same settings.

Test 2- Environment 2

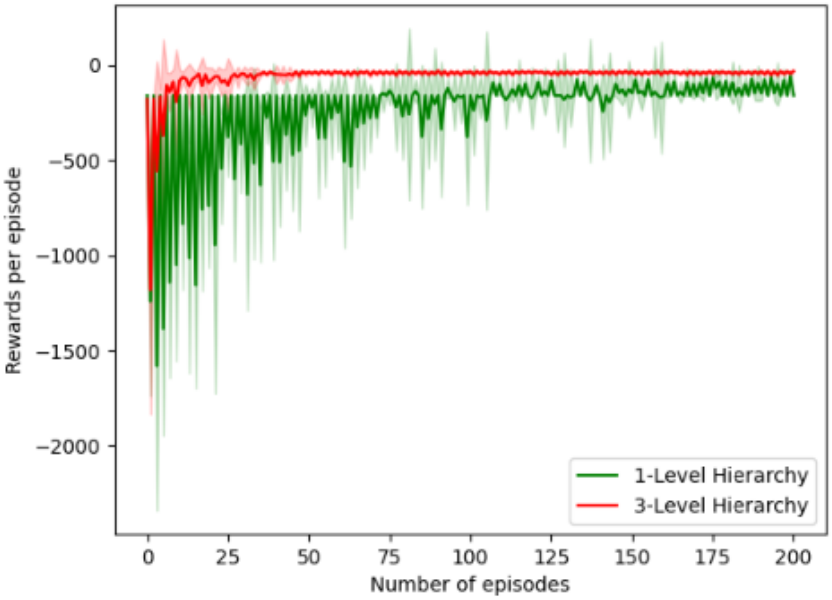

Figure 5.7: Second test in Experiment 2: Agents trained in Environment 2 over the course of 200 episodes.

The 2nd test in Environment 2 behaves similarly to the 2nd test in the first environment (200 episodes), but with the cumulative rewards being smaller per episodes due to the environment being bigger. In this case, our approach learns considerably faster than the flat agent, but in the last episodes, the flat agent performs nearly as good as out approach. For this reason, the third test was carried out, to see if indeed the flat agent continued to perform better and better when the number of episodes increased and manage to outperform the hierarchical approach.

Test 3- Environment 2

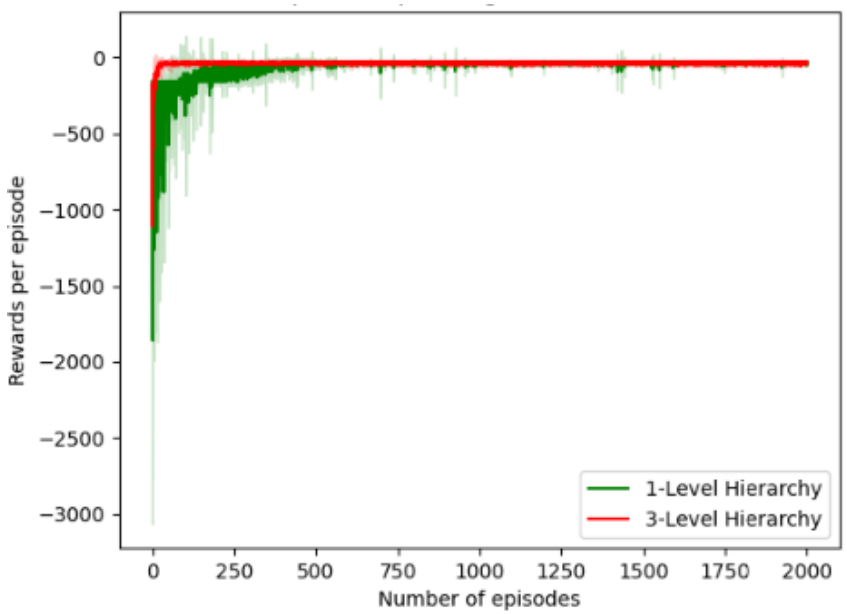

Figure 5.8: Third test in Experiment 2: Agents trained in Environment 2 over the course of 2000 episodes.

The final test in Environment 2 shows that indeed, the hierarchical approach only speeds-up the learning process but does not have a considerable positive impact in the long run (2000 episodes). As in all tests, the HRL approach learned an optimal policy faster than the flat agent, an the standard deviation of the agent was smaller at all times. However, it is showed that when the number of episodes increase, the agents gather a similar amount of reward per episode.

5.3.3 Summary of results

Overall, the different experiments have given us a better insight on how both algorithms behave when changing some settings (episodes and environments). By running experiments in both environments, we have seen the behaviour of each approach (hierarchical and non-hierarchical), and in what circumstances one performs better than the other.

The results have shown that our approach speeds-up the learning process in comparison to the non-hierarchical approach. This could be seen in all tests. Furthermore, it has been seen that our approach is more reliable (less standard deviation) than the flat agent, which shows that the algorithm is more stable and will gather similar rewards when running the same experiments. However, by increasing the number of training episodes, the flat agent achieves an optimal policy that performs better than the hierarchical agent.

These results translate to the ride-sharing setting in the following way. Our approach makes the agent (vehicle) use less episodes to successfully deliver a passenger to a destination. When the vehicle is learning how to pick-up and drop-off the passenger, it will always perform similarly if it is retrained under the same conditions. On the other hand, a vehicle that is learning to perform this task and follows the flat-agent policy, will sometimes deliver the passenger following an efficient path, and in other cases with the same settings, take much more time (high standard deviation). However, by increasing the training episodes, the vehicle that follows the non-hierarchical algorithm will learn to pick-up and deposit passengers using a shorter path than the hierarchical agent (vehicle).

6 Conclusion

In general, the project has achieved all the objectives presented in Section 1.2:

1. Apply a HRL approach to the ride-sharing field. The first environment presented was created in order to simulate the Taxi-v3 environment described in Section 4.2, and the second environment to create a bigger environment with the same components: vehicle, passenger and destination
2. HierQ was modified so the agent could navigate to a sub-goal (passenger) that was in any location in the Grid-World, as opposed to the original implementation, which the suggested sub-goals would always shorten the distance between the agent and the final destination.
3. Finally, it was demonstrated empirically that by applying a Hierarchical Reinforcement Learning approach, the agent could converge to an optimal policy using fewer episodes than the non-hierarchical approach. This implied that the HRL approach used less data samples to converge.

The main point of this project has been that, as explained in Section 2.5, Hierarchical Reinforcement Learning can reduce the time complexity of a task by dividing it in sub-tasks. In our case, this has been achieved by learning the three levels of policies in the hierarchy in parallel, each solving a small problem.

6.1 Future work

There are several topics that a future user could explore and build on top of this project. As it is a ride-sharing environment, one possible addition could be to transform this setting in a multi-agent RL problem. Instead of having one vehicle in the Grid-World (blue square), one could extend this project and enable multiple agents in the same environment. This would be an interesting topic to investigate, as it would be closer to the ride-sharing reality that companies such as Uber and Lyft experience. However, this would require a larger training

process, as not only multiple agents would need to be present, but the number of agents present in the Grid-World would also need to be calculated.

Another interesting topic to study would be to enable the agent to pick-up and drop-off several passengers on the same episode. For now, we extended the original algorithm so the agent could learn to pick-up and deposit passengers around the whole environment, but it was unable to pick-up and deliver multiple passengers per episode. This variation of the agent was studied in this project but finally not implemented, but here are two possible approaches to this problem:

1. As the multi-level hierarchy stored 1 Q-table per goal, one could modify the agent, so it learned how to alternate between Q-tables in the same episode. The challenge would be to train the algorithm for one passenger-destination tuple per episode, and then testing it in an environment where 2 tuples were presented. The agent would need to first select a Q-table to follow (to eventually pick-up a passenger), and then re-evaluate the situation in order to decide whether to follow that same Q-table and drop-off the passenger, or switch to the other Q-table containing the policy to reach the other passenger. Each time a pick-up or drop-off would be performed, the agent would need to re-evaluate the situation and decide what Q-table to follow in order to maximize rewards.
2. The second approach would be to further change the hierarchy architecture of the agent. In this case, the higher-level layer would also contain a Q-table for the setting in where 2 passengers are present in the episode (in addition to the ones for 1 passenger-location tuple). This method would require more training time, as the agent would require an additional $(\text{number-of-square}s)^4$ tables. This indicates all possible combination of two passengers and two destinations anywhere in the Grid-World.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

[2] Matthew M Botvinick, Yael Niv, and Andew G Barto. Hierarchically organized behavior and its neural foundations: a reinforcement learning perspective. *Cognition*, 113(3): 262–280, 2009.

[3] Saeed Vasebi and Yeganeh M. Hayeri. Investigating taxi and uber competition in new york city: Multi-agent modeling by reinforcement-learning, 2020.

[4] Andrew Levy, Robert Platt Jr., and Kate Saenko. Hierarchical actor-critic. *CoRR*, abs/1712.00948, 2017. URL http://arxiv.org/abs/1712.00948.

[5] Uber. Form 10-k. https://www.sec.gov/ix?doc=/Archives/edgar/data/0001543151/000154315120000010/fy2019q410kfinancialst.htm#s90a68fc60ade46349ffbaf5698b1349e. Accessed: 2021-1-15.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017. URL http://arxiv.org/abs/1708.05866.

[7] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[9] Ofir Nachum, Haoran Tang, Xingyu Lu, Shixiang Gu, Honglak Lee, and Sergey Levine. Why does hierarchy (sometimes) work so well in reinforcement learning? *CoRR*, abs/1909.10618, 2019. URL http://arxiv.org/abs/1909.10618.

[10] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *CoRR*, abs/1805.08296, 2018. URL http://arxiv.org/abs/1805.08296.

[11] Leon Bergen, Dzmitry Bahdanau, and Timothy J. O'Donnell. Jointly learning truth-conditional denotations and groundings using parallel attention. *CoRR*, abs/2104.06645, 2021. URL https://arxiv.org/abs/2104.06645.

[12] Izhar Wallach, Michael Dzamba, and Abraham Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery, 2015.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[14] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL https://www.sciencedirect.com/science/article/pii/089360809190009T.

[15] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/f7cade80b7cc92b991cf4d2806d6bd78-Paper.pdf.

[16] Hrushikesh Mhaskar and Tomaso Poggio. Deep vs. shallow networks : An approximation theory perspective, 2016.

[17] M. P. Deisenroth*, G. Neumann*, and J. Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, pages 388–403, 2013. doi: 10.1561/2300000021. URL https://www.nowpublishers.com/article/Details/ROB-021.

[18] Andrew Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, 13, 12 2002. doi: 10.1023/A:1025696116075.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[20] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[21] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. *CoRR*, abs/1711.03938, 2017. URL `http://arxiv.org/abs/1711.03938`.

[22] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey, 2021.

[23] Ashutosh Singh, Abubakr Alabbasi, and Vaneet Aggarwal. A distributed model-free algorithm for multi-hop ride-sharing using deep reinforcement learning, 2019.

[24] Michał Maciejewski and Kai Nagel. The influence of multi-agent cooperation on the efficiency of taxi dispatching. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 751–760, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-55195-6.

[25] Zhiwei Qin, Jian Tang, and Jieping Ye. Deep reinforcement learning with applications in transportation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3201–3202, 2019.

[26] Doina Precup. Temporal abstraction in reinforcement learning, 2000.

[27] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, page 271–278, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602747.

[28] Thomas Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *The Journal of Artificial Intelligence Research (JAIR)*, 13, 12 2000. doi: 10.1613/jair.639.

[29] Craig Boutilier, Ronen Brafman, and Christopher Geib. Prioritized goal decomposition of markov decision processes: Toward a synthesis of classical and decision theoretic planning. *Proc. UAI*, 2, 06 1997.

[30] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, and koray kavukcuoglu. Strategic attentive writer for learning macro-actions. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/c4492cbe90fbdbf88a5aec486aa81ed5-Paper.pdf.

[31] Sainbayar Sukhbaatar, Emily Denton, Arthur Szlam, and Rob Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning. *CoRR*, abs/1811.09083, 2018. URL http://arxiv.org/abs/1811.09083.

[32] Kangenbei Liao, Qianlong Liu, Zhongyu Wei, Baolin Peng, Qin Chen, Weijian Sun, and Xuanjing Huang. Task-oriented dialogue system for automatic disease diagnosis via hierarchical reinforcement learning, 2020.

[33] Xin Wang, Wenhu Chen, Jiawei Wu, Yuan-Fang Wang, and William Yang Wang. Video captioning via hierarchical reinforcement learning. *CoRR*, abs/1711.11135, 2017. URL http://arxiv.org/abs/1711.11135.

[34] Chengshu Li, Fei Xia, Roberto Martin Martin, and Silvio Savarese. HRL4IN: hierarchical reinforcement learning for interactive navigation with mobile manipulators. *CoRR*, abs/1910.11432, 2019. URL http://arxiv.org/abs/1910.11432.

[35] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00052-1. URL https://www.sciencedirect.com/science/article/pii/S0004370299000521.

[36] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *CoRR*, abs/1609.05140, 2016. URL http://arxiv.org/abs/1609.05140.

[37] Jean Harb, Pierre-Luc Bacon, Martin Klissarov, and Doina Precup. When waiting is not an option : Learning options with a deliberation cost. *CoRR*, abs/1709.04571, 2017. URL http://arxiv.org/abs/1709.04571.

[38] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, page 1043–1049, Cambridge, MA, USA, 1998. MIT Press. ISBN 0262100762.

[39] George Konidaris and Andrew Barto. Skill chaining: Skill discovery in continuous domains. 01 2010.

[40] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *CoRR*, abs/1703.01161, 2017. URL `http://arxiv.org/abs/1703.01161`.

[41] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017. URL `http://arxiv.org/abs/1707.01495`.

[42] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1312–1320, Lille, France, 07–09 Jul 2015. PMLR. URL `http://proceedings.mlr.press/v37/schaul15.html`.