UNIL | Université de Lausanne
HEC Lausanne

Master Thesis

A Reinforcement Learning Approach To Food Delivery and Ride-Sharing

December 16, 2020

Author:
Alexandre Schroeter

Supervisors:
Prof. Olivier Gallay
Dr. Marc-Olivier Boldi

Master in Management,
Orientation Business Analytics
UNIL

HEC Lausanne

1 Abstract

This work aims to present an extension of the taxi problem (Diettrich, 2000) with three algorithms of Reinforcement Learning (RL). The original taxi problem consists in driving one passenger from one of four possible origins to one of four possible destinations. In the present case, this problem is extended to two passengers/parcels and five different origins/destinations creating 22,500 different states. The initial environment is also customized by adding horizontal walls. Three different agents are used and tested; Q-learning, a tabular method, Deep Q-Network (DQN), an approximate method and an Actor-Critic model (AC), a policy-based approach. We first present reinforcement learning overall as well as the detailed theory behind each algorithm. We also present the hyperparameter tuning approaches; the grid search, the random search, the Bayesian optimization, and the evolutionary algorithm. We then explain the customization process of the environment. Each agent's implementation is thoroughly described. We manually implement the Q-learning algorithm while we use the Stable-Baselines library for the DQN and the PyTorch library to build the Actor-Critic model. An important issue is also raised; the complexity associated to increasing the number of passengers and destinations. In fact, adding a third passenger would increase the number of states from 22,500 to 675,000. This characterizes our whole approach to this multi-parcel delivery problem. We then use the Optuna library for the hyperparameter tuning process. Q-learning, DQN, and AC have respectively 30, 6, and 5 different combinations tested. We find that Q-learning mean reward per episode over 200,000 episodes is 21.70. DQN mean episodic reward over 100 episodes is slightly better with 22.46. Surprisingly though, the mean episodic reward of AC is worse, with 15.25 over 100 episodes. This may be due to the limited number of combinations tested during the hyperparameter tuning. However, when we identify the opportunities for ride-sharing, we find that the Q-learning algorithm does not develop any specific strategy for the cases where the two passengers/parcels could be carried at the same time. Instead, Q-learning transports the parcels/passengers one after another. On the other hand, AC has developed such a strategy and is, therefore, more promising. Finally, we conclude by discussing the limitations of the different RL algorithms.

Contents

2 Introduction

Major steps in machine learning and computational capabilities have opened new possibilities in recent years. Deep learning saw strong growth and received increased attention in the early 2010s. From near-human-level image classification to a superhuman Go playing ability, deep learning has experienced many successes in the past decade. In the meantime, the sharing economy emerged and new business models such as Uber or airbnb were created. It is estimated that the sharing economy will increase from $ 14 billion in 2014 to $ 335 billion by 2025. From an economic point of view, route optimization and ride-sharing bring numerous benefits. They can alleviate congestion by reducing the number of cars on the road and appear as an imperative to reduce energy usage and pollution. Thus, they offer an interesting way to improve socio-economic benefits and reduce negative externalities. In the present work, we aim to use a particular subfield of machine learning, reinforcement learning, to try to optimize the path of a food delivery man. This problem is similar to that of a taxi having to drive passengers from a given origin to a given destination. The taxi is instead a delivery man, the origin is the restaurant, and the destination is the final customer. Common methods to answer this type of problems include finding the nearest driver to serve a passenger or using queueing strategies with the principle of first-come-first-serve. Many traditional approaches rely on heuristics (e.g., low profitability regions, low vacancy rate) based on aggregate statistics from the data to myopically optimize revenue. Despite being easy to implement, these methods are naturally uncoordinated and tend to prioritize immediate passenger satisfaction over global supply utilization. Moreover, ride-sharing and food delivery are inherently challenging because the environment is dynamic. At each step, a decision-making process occurs. The objective function evolves based on the action the driver must make and based on uncertainties such as the vehicle's location, the restaurants' locations, and the clients' destinations. Therefore, a model-based approach is unable to consider all the intricacies involved in such an optimization problem and is unlikely to yield good results. A flexible approach seems better. Consequently, this work aims to present, optimize, and apply three different reinforcement learning algorithms to optimize the path of a food delivery man and to present and reflect on the results of this approach.

3 Theory

3.1 Introduction

This section aims to summarily present the theory used throughout this work. Furthermore, the different algorithms applied for this purpose are also detailed and explained.

Reinforcement learning is a class of unsupervised machine learning algorithms which act as so-called agents. An agent operates in an environment which can be priorly known or unknown. The environment is made of everything outside the agent. A reinforcement learning system comprises four different elements: a policy, a reward, a value function, and optionally a model of the environment.

Reinforcement learning problems are theoretically idealized as Markov Decision Processes (MDPs). MDPs are a classical formalization of sequential decision-making where actions, made by the agent, do not only influence the immediate rewards but also subsequent situations, states, and rewards. Therefore, MDPs require that the agent makes a trade-off between an immediate reward and discounted future rewards.

3.2 Agent Environment Relationship

The agent and the environment interact with one another. These interactions happen at time $t \in \mathcal{T}$ in the discrete case. At each time step t, the agent receives some representation of the environment state's $S_t \in \mathcal{S}$. Based on this representation, the agent selects an action $A_t \in \mathcal{A}(s)$, from the set of all possible actions available at time t to the agent. One time step later, partly because of this action[1], the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$ and enters a new state S_{t+1}. The reward can either be positive or negative, depending on how the problem is formulated.

Thus, the MDP and the agent trajectory looks like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots$$

[1]Note: Partly because, if the transition from one state to the next is deterministic, then the transition is only the result of the action. In the stochastic case, when the agent selects an action, it may not reach the desired state because the transition is stochastic and does not only depend on the selected action.

In the present case, each interaction between the agent and the environment can be divided into subsequences, as illustrated by Figure 1. All these interactions create an episode. When the episode is over, when the agent has reached his goal, been stuck or terminated, the agent is reinitialized in a new stage and a new episode begins.

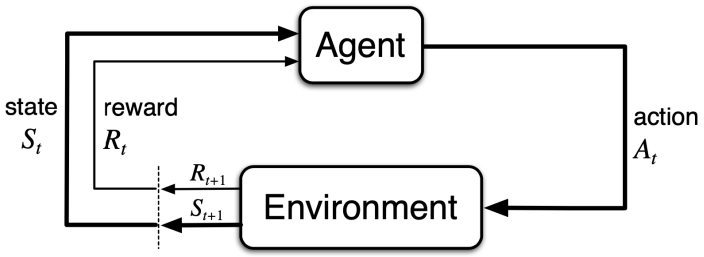


Figure 1: The agent-environment interaction in a Markov decision process.

Source: Sutton, Richard S and Barto, Andrew G, *Reinforcement learning: An introduction*, page 48, 2018, MIT press

The agent is trying to select actions to get the highest sum of discounted rewards. The expected discounted return is defined as such:

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

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate.

To achieve the goal of the highest sum of discounted rewards, the agent tries to learn an optimal behaviour, a policy π, about how to behave when it finds itself in a given state.

Formally, a policy is "*a mapping from the states to probabilities of selecting each possible action*". If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. [2] To estimate how good it is for it to be in a given state, and to design an "optimal" policy, the agent uses two functions; the value function and the action-value function.

[2] A slightly different notation is used throughout this work. Random variables are denoted with capital letters while their instantiations are written in lower case. For example, the state, action and reward at time step t are written as S_t, A_t, and R_t, but the values they take are s, a, and r respectively. It is also natural to use lower case for value functions (e.g, v_π) while the tabular estimates, such as in Q-learning in Section 3.7, are written in upper case (e.g., $Q_t(s, a)$). Also, s_{t+1}, a_{t+1}, and s' and a' are used interchangeably.

3.3 Value And Action-Value Functions

The state-value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when one is in state s and follows policy π afterwards. For MDPs, we can define v_π formally:

$$v_\pi(s) \doteq \mathbb{E}[G_t|S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right]$$

The action-value is the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking action a and thereafter following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]$$

We define the optimal state-value function v_* and the optimal action-value function q_* as such:

$$v_* \doteq \max_\pi v_\pi(s)$$

$$q_* \doteq \max_\pi q_\pi(s, a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

3.4 Policy Evaluation, Improvement And Iteration

3.4.1 Policy Evaluation

First, it is necessary to be able to evaluate an arbitrary policy π. Using Dynamic Programming, it is possible to write:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π and $p(s', r|s, a)$ is the probability of transitioning to state s' with reward r from state s with action a.

The expectations are subscripted by π to indicate that they are conditional on π being followed. We then sum over all possibilities to get an expected value.

3.4.2 Policy Improvement

The reason to compute the value function for a given policy lies in the improvement of this policy. One is interested to know whether this policy can be improved and if so, how it should be improved. Should we consider different actions of the current policy ?

Let us define π as the current policy under evaluation and π' as a changed policy where some actions are different than those of π.

First, we must consider the action-value function:

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

Thanks to the policy improvement theorem, we can state that, given a policy π and its value function $v_\pi(s)$, we can easily evaluate a change in the policy (resulting in a new policy π') at a single state s to a particular action a:

$$v_\pi(s) \leq q_\pi(s, \pi'(s)) = v_{\pi'}$$

If we extend this to all states and all possible actions, selecting at each state the action that appears best according to the action-value function $q_\pi(s, a)$, we get:

$$\begin{aligned} \pi'(s) &\doteq \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

Policy improvement is, therefore, the process of making a new policy which improves an

original policy by making it greedy[3] with respect to the value function of the original policy.

3.4.3 Policy Iteration

The process of finding an optimal policy is called policy iteration. Once a policy π has been improved and yields a better policy π' using the value function v'_π, it can be improved iteratively by yielding a better policy π'' using v''_π. Therefore the process works as follows:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \dots \pi_* \xrightarrow{E} v_{\pi_*}$$

where $\xrightarrow{E}$ and $\xrightarrow{I}$ are respectively the evaluation and improvement procedures.

3.5 On-Policy And Off-Policy Methods

One of the dilemmas an algorithm faces is the trade-off between exploration and exploitation. While the algorithm is trying to learn action-values based on subsequent optimal behaviour, it still needs to behave non-optimally to explore the action space to find the optimal actions. To address this issue, two methods exist; the on-policy and off-policy approaches.

The on-policy approach is a compromise and learns action values for a near-optimal policy which continues to explore. On-policy learning algorithms are algorithms which evaluate and improve the same policy which is being used to select actions. This means that we will try to evaluate and improve the same policy that the agent is already using for action selection. Another approach, the off-policy method, consists of two different policies; a target policy π and a behaviour b. A target policy π is being learnt and will become the optimal policy. In the meantime, the behaviour policy b is used to explore the environment by interacting with it and thus generates new behaviour for the target policy π. On-policy learning is a special case of off-policy learning where $\pi = b$. Off-policy learning algorithms evaluate and improve a policy which is different from the policy used for action selection.

[3] In the Reinforcement Learning context, greedy means to choose the action believed to give the highest expected reward.

3.6 Algorithm Classes

In Reinforcement Learning, two broad families of algorithms exist; tabular solution methods and approximate solution methods. If the state space and action space are small enough, the value functions can be represented as arrays, or tables. In such a case, exact solutions for the optimal value function and the optimal policy can be determined. This contrasts with the approximate methods where only approximate solutions can be found. In many tasks, the state space is enormous. In these cases, it is impossible to find an optimal value function and an optimal policy. Instead, we want to find a good approximation of the optimal solution.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states, it is necessary to generalize from previous encounters with different states, that are in some sense similar to the current one. In other words, the key issue is that of generalization. The kind of generalization we require is often called function approximation because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function.

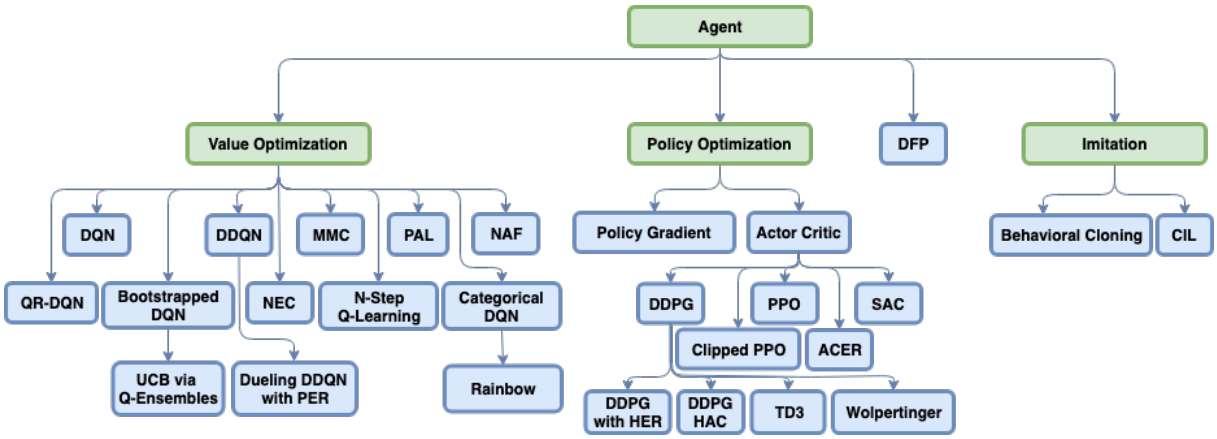


Figure 2: Classes of algorithms in the approximate methods family

Source: https://nervanasystems.github.io/coach/selecting_an_algorithm.html

Figure 2 gives a visual representation of the different subfamilies in the approximate methods family, the one useful for large state spaces. Algorithms can be classified into 3 subfamilies; value optimization, policy optimization, and imitation.

Value optimization algorithms or value-based approaches aim to estimate the value function using function approximators (neural networks). Intuitively, the value function $v_\pi(s)$ measures how good it is to be in a specific state following policy π. Their output is an approximation function. Value optimization algorithms expect to receive examples of the desired input-output behaviour of the function they are trying to approximate. These methods often use supervised learning to do so. Achieving the best expected discounted return remains the goal.

Policy optimization algorithms are different from value-based approaches because they do not require action-values. In the latter, a value function is learnt from which a policy is derived. In policy gradient methods, a parametrized policy that can select actions without consulting a value function is learnt directly. We consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to maximize performance, so their updates approximate gradient ascent in $J(\cdot)$.

Imitation algorithms are a class of algorithms which perform imitation learning. Imitation learning is a means of learning and developing new skills from observing these skills performed by another agent. It is useful when it is easier for an expert to demonstrate the desired behaviour rather than to specify a reward function which would generate the same behaviour or to directly learn the policy.

3.7 Q-learning

Temporal-Difference (TD) learning is a class of algorithms where these methods can learn directly without a model of the environment. TD methods update estimates based on other learned/previous estimates, without waiting until an episode is complete to update all estimates. This is called *bootstrapping*.

One of the most used algorithms in this class is the Q-learning algorithm (Watkins, 1989) also called Off-Policy TD Control. The update of the Q-function is defined as such:

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

The data used to train the agent is collected through interactions with the environment by the agent itself (compared to supervised learning where you have a fixed dataset for instance). This algorithm requires that a table, the Q-table, include all states. This can be an issue if the cardinality of states is high. The full algorithm is detailed in Section 8.1.

3.8 Deep Q-Network (DQN)

Q-learning is an algorithm useful when the state space is small. When this is not the case, then the algorithm may be slow to converge because it estimates precisely the state-value and the action-value and updates them continuously.

When the state space is too large, approximating the state-value and the action-value becomes necessary. To this end, another subfield of Machine Learning, supervised learning is useful for function approximation.

In the tabular case, measuring the prediction quality continuously is not necessary because the learned value function could converge easily to the true value function. This is not the case in approximate methods such as DQN.

In theory, we have more states than parameters associated with the features used to approximate the state-value and the action-value functions. There is thus a tradeoff to make where we must say which states are the most important to us. We define a state distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$, where μ represents how important the state is. The error in the state approximation is defined as the squared difference between the approximate value $\hat{v}(s, \boldsymbol{w})$ and the true value $v_\pi(s)$. Thus, the error in the value approximation is defined as :

$$\overline{\text{VE}}(\boldsymbol{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2$$

where $\boldsymbol{w} \in \mathbb{R}^d$ is the vector of parameters associated with the parameterized functional form used for approximation of the value function.[4]

One of the most useful classes of algorithms used to approximate functions non-linearily

[4]In the case of Artifical Neural Networks (ANN), most research papers use $\boldsymbol{\theta}$ to denote the weight vector of the ANN. However, to avoid confusion with the policy-based approach, we use $\boldsymbol{w}$.

is Artificial Neural Networks (ANN) most commonly known as Deep Learning. Deep learning has the capability of self-extracting useful features (automation of the feature design process) with little human involvement. Figure 3 is a representation of how ANN can be applied in Reinforcement Learning.

In the case of DQN (Mnih et al., 2013), the approximation of the value functions is performed non-linearily. To do so, stochastic gradient descent (SGD) methods are necessary. In this case, the weight vector $\boldsymbol{w}$ is a column vector with d real-valued components $\boldsymbol{w} \doteq (w_1, w_2, \ldots, w_d)^T$ and the approximate value function $\hat{v}(s, \boldsymbol{w})$ is a differentiable function with respect to $\boldsymbol{w}$ for all $s \in \mathcal{S}$. In general, there is no $\boldsymbol{w}$ which perfectly approximates all the known or unknown states. To this end, it is necessary to adjust $\boldsymbol{w}$ by updating it to lower the prediction error:

$$\begin{aligned} \boldsymbol{w}_{t+1} &\doteq \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w})\right]^2 \\ &= \boldsymbol{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w})\right] \nabla \hat{v}(S_t, \boldsymbol{w}_t) \end{aligned}$$

where α is a positive step-size parameter and $\nabla f(\boldsymbol{w})$ is the vector of partial derivatives with respect to the components of the vector $\boldsymbol{w}$:

$$\nabla f(\boldsymbol{w}) \doteq \left(\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \frac{\partial f(\boldsymbol{w})}{\partial w_2}, \ldots, \frac{\partial f(\boldsymbol{w})}{\partial w_d}\right)^T$$

As the basis of the DQN is Q-learning, a bootstrapping method, this raises an issue. Our estimates of the value function $v_\pi(S_t)$ is noisy and thus we have no guarantee that the SGD method will converge to a local optimal approximation of $v_\pi(S_t)$. This is due to our using of a bootstrapping estimate of the value function which depends on the weight vector $\boldsymbol{w}$. Therefore the independence between $v_\pi(S_t)$ and $\boldsymbol{w}$ can no longer be assumed and this will not lead to a true gradient-descent method (Barnard, 1993).

The semi-gradient method for the Q-learning algorithm approximation is :

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \Bigg[\underbrace{R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \boldsymbol{w}_t)}_{\text{target}} - \hat{q}(S_t, A_t, \boldsymbol{w}_t) \Bigg] \underbrace{\nabla \hat{q}(S_t, A_t, \boldsymbol{w}_t)}_{\substack{\text{gradient of} \\ \text{current predicted} \\ \text{Q-value}}} \tag{2}$$

If we use SGD, we move closer to the target but the target also moves. This does not happen in a supervised learning task. To avoid this problem, we use semi-gradient methods. These take the effect of changing the weight vector $\boldsymbol{w}_t$ on the estimate into account, but they ignore the effect on the target.

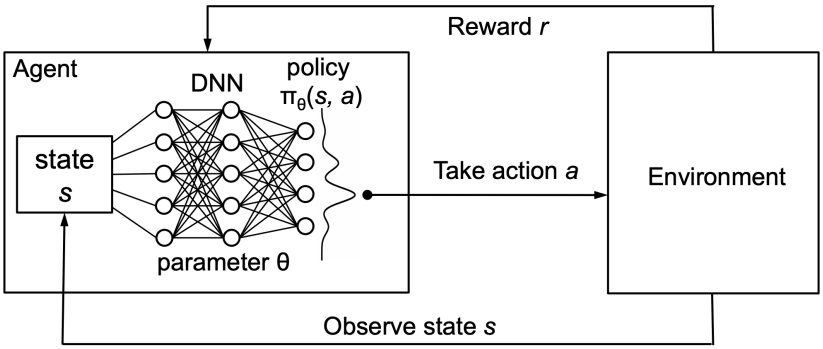


Figure 3: Reinforcement Learning with policy represented via Deep Neural Network

Source: https://www.novatec-gmbh.de/en/blog/deep-q-networks/

3.8.1 Deep Q-Network With Experience Replay

The motivation behind the use of the Q-learning algorithm in a neural network is two-fold. First, Q-learning is a simple algorithm compared to other methods and this reduces the time for learning. Second, one of the main characteristics of DQN, detailed in Section 8.2, is the use of *experience replay* and of a *target network* which decrease instability. Most learning algorithms make the strong assumption that the data samples are independent. This is not the case in reinforcement learning. The agent frequently encounters sequences of highly correlated states.

To prevent correlation, using experience replay is useful. At each time step, the agent's experiences, $e = (s_t, a_t, r_t, s_{t+1})$ are stored in a memory, also known as a buffer of arbitrary capacity N. During the inner loop of the algorithm, we perform Q-learning/minibatch updates of size N_b, where we randomly sample experiences, following a uniform distribution, from the memory $\mathcal{D}$.

The target network is the same as the online network except that its parameters $\boldsymbol{w}^-$ are copied every τ steps from the online network.

This method has several advantages. It increases data efficiency by using potentially

each step of experience in many weight updates. Secondly, it increases learning efficiency by removing correlations between states and reducing the variance of the updates. Learning from two highly correlated states is less useful. Third, by using experience replay, the behaviour distribution is averaged over many previous states. This decreases the oscillations and divergence of the parameters. One important requirement for this approach is to use an off-policy algorithm. To this end, Q-learning is very appropriate. One disadvantage of this method is the finite size of memory. The buffer does not differentiate between important transitions as it always overwrites recent transitions over older ones. To counter this, *prioritized experience replay* may be useful. The key idea is to increase the replay probability of experience tuples which have high expected learning progress based on a reasonable proxy of the magnitude of the transition; the TD error δ.

3.8.2 Double Deep Q-Network With Experience Replay

The main issue with Q-learning and Deep Q-Network is their tendency to learn unrealistically high action values and lead to a maximization bias. The cause of this is the way Q-learning was designed, using the max operator. This is the same case for a simple DQN (without extension) as illustrated below, with Y_t^{DQN} being the target for the update:

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}_t)$$

As it includes a maximization over estimated action values, it tends to prefer overestimated values. Van Hasselt et al. (2015) proved that such overestimations are common and not uniform across states and actions when using the Q-learning algorithm.

Overestimation and bootstrapping can propagate the wrong relative information about the state values, leading to a decrease in the quality of the learned policy.

To prevent this, Van Hasselt (2010) had the idea of implementing Double Q-learning with a neural network. Double Q-learning decomposes the max operation in the target, Y_t, into action selection and action evaluation. Two value functions are learned by assigning each experience randomly to update one of the two functions, such that there are two sets of weights, $\boldsymbol{w}$ and $\boldsymbol{w}^-$. The Double Q-learning error can be written as

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{w}_t); \boldsymbol{w}_t^-) \tag{3}$$

Equation (3) means that we still **estimate** the value of the greedy policy according to the set of weights, $\boldsymbol{w}$, but we use $\boldsymbol{w}^-$ to **evaluate** the value of the policy. Both $\boldsymbol{w}$ and $\boldsymbol{w}^-$ are symmetrically interchangeable, as in Double Q-learning.

Thus, this approach stabilizes the learning process by decreasing the variance of the estimation.

3.8.3 Dueling Network Architecture

Both algorithms presented until now have used the same standard architecture, with one final layer estimating the action-values. A complementary approach is to build a neural network architecture more suitable for model-free reinforcement learning; the dueling architecture. This architecture consists of two streams which represent the value $V(s)$ and advantage $A(s, a)$ functions, as illustrated by Figure 4. Finally, the two streams are combined into an aggregating layer to produce an estimate of the state-action $Q(s, a)$ value function Q. The intuition behind the dueling architecture is that it can learn which states are valuable or not, without the need to learn the effect of each action on these states.

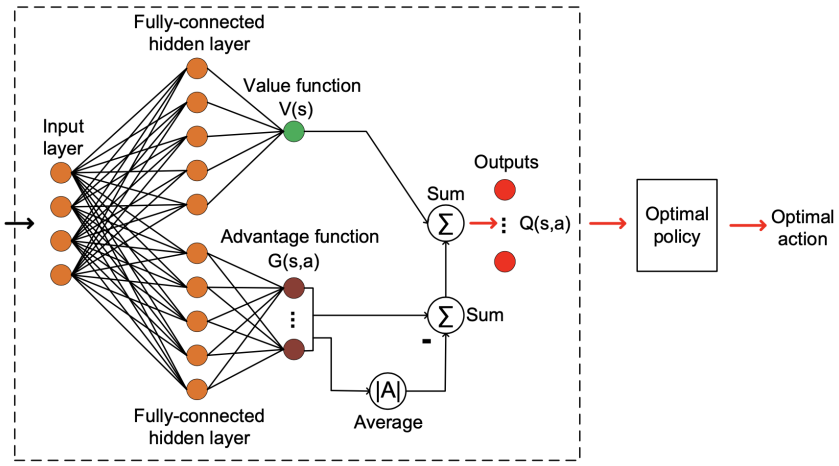


Figure 4: Deep dueling neural network architecture

Source: Van Huynh, Nguyen and Nguyen, Diep N and Hoang, Dinh Thai and Dutkiewicz, Eryk, *Jam Me If You Can: Defeating Jammer With Deep Dueling Neural Network Architecture and Ambient Backscattering Augmented Communications*, IEEE Journal on Selected Areas in Communications, volume 37, number 11, pages 2603–2620, 2019, IEEE

The first stream of fully-connected layers outputs a scalar $V(s; \boldsymbol{w}, \boldsymbol{\beta})$. The output of the second stream, the advantage is $A(s, a; \boldsymbol{w}, \boldsymbol{\alpha})$. $\boldsymbol{w}, \boldsymbol{\beta}, \boldsymbol{\alpha}$ are the parameters associated with

the neural network (excluding the two top layers), the parameters associated with the first stream layer, and the parameters associated with the second stream layer.

As before, the $Q_\pi(s,a)$ and V_π are defined as:

$$Q_\pi(s,a) = \mathbb{E}[R_t|S_t = s, A_t = a, \pi]$$

$$V_\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q_\pi(s,a)]$$

The other important quantity is the advantage function:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

This is a relative measure of the importance of each action compared to the others.

Finally, we combine the two streams into the final layer, $Q(s,a;\boldsymbol{w},\boldsymbol{\alpha},\boldsymbol{\beta})$:

$$Q(s,a;\boldsymbol{w},\boldsymbol{\alpha},\boldsymbol{\beta}) = V(s;\boldsymbol{w},\boldsymbol{\beta}) + \left(A(s,a;\boldsymbol{w},\boldsymbol{\alpha}) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\boldsymbol{w},\boldsymbol{\alpha})\right) \tag{4}$$

The way Equation (4) is defined addresses the issue of identifiability to recover V and A uniquely.

In their paper, Wang et al. (2016), showed that, when the state space is large, the dueling architecture performs better than a traditional DQN and thus leads to much faster convergence.

Contrary to the single-stream architecture where only one action-value is updated, the whole value stream V is updated for every update of the Q values.

Since the output of the dueling network is a Q function, it can be trained with many existing algorithms, such as DQN or SARSA. Moreover, it can be combined with methods described previously such as experience replay or Double Deep Q-Network.

The Deep Q-Network family belongs to value-estimation approaches. It has the advantage of being mathematically simpler and easy to implement. The action-values are estimated and a policy τ is developed based on them. Therefore, the policy is not the objective per se but is indirectly looked for when we choose the best action-values. A value-based

method could be complemented with a policy-based method. These methods have several advantages; they have better convergence properties, they are more effective in high-dimensional spaces, and they can learn stochastic policies.

3.9 Actor-Critic Model

3.9.1 Policy-Based Approach

A policy-based method learns a policy directly, without the need of a value function. A value function may still be used to learn the policy parameter but is not required for action selection. We define $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ as the policy's parameter vector. Given that the environment is in a state s at time t with parameter $\boldsymbol{\theta}$, the probability that action a is taken is:

$$\pi(a|s, \boldsymbol{\theta}) = \mathbb{P}(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$$

Policy-based methods learn the policy parameters based on the gradient of a given scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter $\boldsymbol{\theta}$.

As these methods want to maximize performance, so their update approximate gradient ascent in $J(\cdot)$ is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to $\boldsymbol{\theta}_t$.

The policy can be parametrized in any way, as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters, meaning, as long as $\nabla\pi(a|s, \boldsymbol{\theta})$ exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ and $\boldsymbol{\theta} \in R^{d'}$. To make sure the policy keeps exploring, we require that it never becomes deterministic, that is $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$ for all $s, a, \boldsymbol{\theta}$.

For an environment where the action space is discrete and not too large, a common approach of policy parametrization is to form parametrical numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair. An exponential soft-max distribution in action preferences is well-suited for this task:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$$

In the episodic case, the performance metrics is defined at the start state s_0 of an episode:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0) = \mathbb{E}\left[\sum_{t\geq 0}[\gamma^t r_t|\pi_{\boldsymbol{\theta}}]\right] \tag{5}$$

where $v_{\pi_{\boldsymbol{\theta}}}(s_0)$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$.

Therefore, the most optimal parameter vector $\boldsymbol{\theta}^*$ will maximize the expected reward of Equation (5) following the policy such as:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

3.9.2 Differences Between Value-Based And Policy-Based Methods

In value-based approaches, we derive the policy based on the estimates generated by an optimal (well trained) value function. Although the value function was stochastic, the so implied policy in most of the implementation of value-based approaches has been mostly deterministic. This is because the policy in the case of value estimation approaches suggests a single action. This action could either be the best action suggested by the estimation function or any random action, but it lacked the suggested probability distribution for selecting across different actions. On the contrary, in the case of policy approximation approaches, since the policy itself is parameterized, it is stochastic. This essentially means that for a given state, the policy may have varying probabilities of choosing different actions instead of choosing the single most optimal action. So, the policy-based approaches would essentially draw samples from this stochastic policy to refine their estimate of the policy parameter vector $\boldsymbol{\theta}$ in a direction (as in the gradient) to optimize the policy which would subsequently enable the agent that follows such policy to accumulate maximum cumulative reward. In the case of policy-based approaches, these can stochastically suggest multiple actions with appropriate probabilities.

One implementation of the policy-based method is the REINFORCE algorithm (see Appendix (3)). It is a Monte Carlo algorithm because it uses the complete return from time t, which includes all future rewards up until the end of the episode. However, as it is a Monte Carlo algorithm, it has high variance and slower learning. One reason for such high variance is the form in which the rewards are used in REINFORCE. In

REINFORCE absolute rewards are used, and with each experiment of Monte Carlo, the rewards may vary a lot. This leads to a very high variance in the so obtained gradient. This high variance is mainly due to the fact that we are not able to deterministically and specifically identify which actions are attributed to the reward in a given trajectory. To address these shortcomings, including a baseline scenario can help one understand how much an action is better or worse. This is a similar idea as in Dueling DQN where the advantage function helps identify the relative advantage of a given action in a particular state over the base value of the state. As we do not want to go into the calculation of the state value $V(s)$ such as in the DQN case, one baseline could be a constant moving average of the cumulative future discounted rewards received from all the trajectories which pass through this particular state. Nonetheless, REINFORCE still has a wide variance issue in the gradient approximation. In fact, value optimization and policy optimization can complement each other. The weakness of one is almost the strength of the other. Value-estimation methods have less variance and fewer samples while policy-based methods can deal with larger spaces and provide a stochastic policy.

3.9.3 Design Of The Actor-Critic Method

The method, detailed in Subsection 8.3.2, uses both an actor and a critic (Figure 5). The actor has a policy-based approach and applies a policy to take actions. On the other hand, the critic is value-based and provides feedback to the actor on previous actions. First, the actor interacts with the environment by taking an action and hence has a stochastic policy. Then, the resulting reward is sent to the critic which processes it by correcting and updating its

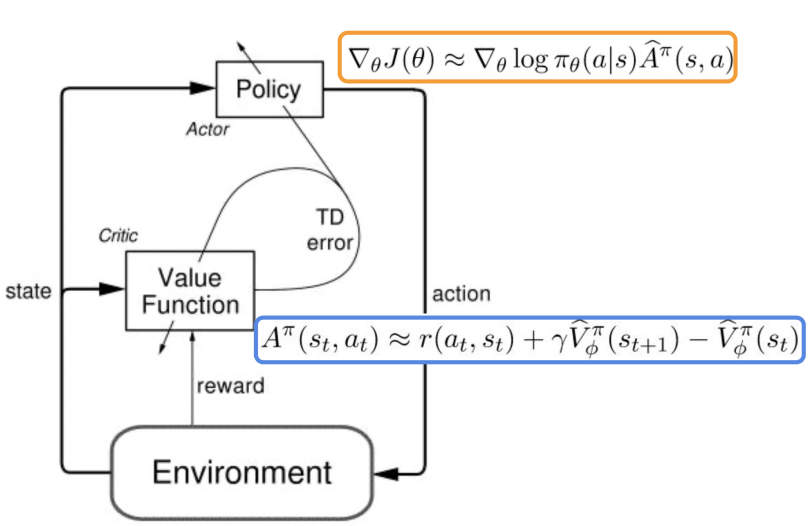


Figure 5: Actor-Critic algorithm principle

Source: https://www.slideshare.net/zhihua98/actor-critic-algorithm?from_action=save

own estimates. Finally, the critic gives feedback to the actor with a corrected value of the estimates. This helps the actor update itself continuously during training. The critic's value estimates serve as a baseline for the actor to update its policy using the policy-gradient approach, just like in the REINFORCE case. Also, even if the reward is not sent directly to the actor, the new state s' is.

The error of the critic (for a one-step update) between the subsequent state values estimates is computed using the instantaneous reward and discounted state value of subsequent state:

$$\delta = R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S', \boldsymbol{w})$$

where γ is the discount rate and $\hat{v}$ the state-value estimator function parametrized by weight vector $\boldsymbol{w}$. We update weight vector $\boldsymbol{w}$ at each step too:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S, \boldsymbol{w})$$

Just like in Q-learning or DQN, α is the learning rate. Finally, the policy estimation function of the actor needs to be updated too by updating the parameter vector $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$$

Two neural networks are required, one for the actor and another for the critic. As both require state inputs, it is possible to improve the computations efficiency by sharing the same architecture for both networks (Figure 6). This is also a way to make sure that the interpretation of the received state and the coordination is common for both the actor and the critic.

For a discrete action space, the actor-network needs to end in a SoftMax activation layer and the critic-network ends with a node with a linear activation function.

As we saw previously, using the advantage as a baseline can reduce the variance in the actor's gradients and increase the learning speed. It is therefore possible to slightly modify the critic and make it generate a (bootstrapped) advantage estimate in place of the state-value estimate. This estimate is then sent to the actor.

Training such models on environments with a large state space can hinder performance. Having multiple asynchronous agents learn in parallel across different instances of the environment can improve this issue as shown in Figure 7.

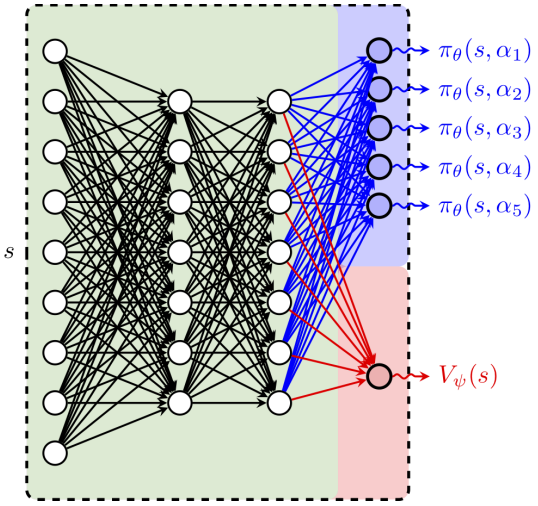


Figure 6: Actor-Critic model shared architecture

Source: https://montrealartificialintelligence.com/academy/

In such architecture, a global centralized network parameter server stores the parameters for both the actors and the critics. Any new agent copies the current values of the parameters from the global networks parameter server and updates their copy of the parameters while training with their own instance of the environment independently.

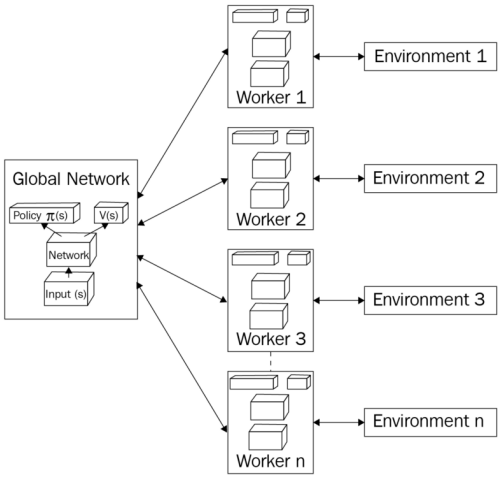


Figure 7: Asynchronous Advantage Actor-Critic Model (A3C)

Source: http://www.henrypan.com/blog/reinforcement-learning/2019/02/27/a3c-rl-layments-explanation.html

After some fixed steps, or reaching the terminal state, the agents merge their updates with the global parameters of the centralized parameter server, then copy the new updated global network parameters, and resume interacting with their respective instance of the environment. Since each agent has their own copy of the environment, essentially each agent is working on different and uncorrelated states (from different instances of the same environment class), and thus, the subsequent global updates are uncorrelated. This brings stability in the training and obviates the need for provisioning very large memory for experience replays for each agent individually. We call this approach Asynchronous Advantage Actor-Critic model (A3C).

3.10 Hyperparameter Tuning

From Q-learning to the Actor-Critic model, every method has several parameters to be optimized, notably α, the step-size, γ, the discount rate, or ε, the proportion of actions chosen randomly in Q-learning and DQN. Moreover, large architectures such as neural networks increase the difficulty of finding optimal solutions. Hyperparameter tuning is one of machine learning's most cumbersome tasks but it does have importance. Hence, this requires a systemic approach. The goal is to find a set of hyperparameters that yields the highest expected discounted reward.

3.10.1 Grid Search

Grid search requires that we arbitrarily choose a set of values for each variable $(L^1, \ldots, L^K)$. The set of trials is then created by combining each value of a given variable with the values of other variables. The number of trials is thus $S = \Pi_{k=1}^{K} |L^{(k)}|$. This approach can easily suffer from the curse of dimensionality if the number of values and/or variables is too high.

3.10.2 Random Search

Random search is a simple and popular method for hyperparameter tuning. It is model-free and can serve as a robust baseline against more complex methods. Hyperparameter configurations $\boldsymbol{\lambda}$ are sampled randomly from a specified hyperparameter space $\boldsymbol{\Lambda}$. At the end of the process, the best observed hyperparameter combination $\boldsymbol{\lambda}^*$ is chosen.

3.10.3 Bayesian Optimization

Bayesian optimization (BO) is a sequential model-based approach to optimize "black-box" functions $f(\boldsymbol{\lambda})$, $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$. Being data-efficient, BO is particularly suitable for hyperparameter optimization. BO specifies a probabilistic prior model over the unknown function f and applies Bayesian inference to compute a posterior distribution over f given the previous observation $\{\boldsymbol{\lambda}, y_i\}_{i=1}^{t}$. The posterior distribution is used to build an acquisition function. By maximizing the acquisition function, one can decide the next query point $\boldsymbol{\lambda}_{t+1}$. One common surrogate model for $f(\boldsymbol{\lambda})$ is a Gaussian Process (GP). Assuming normal observation noise, $\Psi(\boldsymbol{\lambda}) \sim \mathcal{N}(f(\boldsymbol{\lambda}), \tau^2)$ creates a natural framework for uncertainty in observations. Given a collection of observations

$\mathcal{C} = \{(\boldsymbol{\lambda}_i, \boldsymbol{\Psi}_i)\}_{i=1}^n$ and a Gaussian Process prior, the posterior distribution at hyperparameter setting as $f(\boldsymbol{\lambda}|\mathcal{C}) \sim \mathcal{N}(\mu_n(\boldsymbol{\lambda}, \sigma_n^2(\boldsymbol{\lambda})))$.

If we assume a zero prior, the posterior mean and variance are given by:

$$\mu_n(\boldsymbol{\lambda}) = \boldsymbol{k}(\boldsymbol{\lambda})^T(\boldsymbol{K} + \tau^2\boldsymbol{I})^{-1}$$
$$\sigma_n^2(\boldsymbol{\lambda}) = k(\boldsymbol{\lambda}, \boldsymbol{\lambda}) - \boldsymbol{k}(\boldsymbol{\lambda})^T(\boldsymbol{K} + \tau^2\boldsymbol{I})^{-1}\boldsymbol{k}(\boldsymbol{\lambda})$$

where $\boldsymbol{k}(\boldsymbol{\lambda})$ is the vector of covariances with all previous observations and $\boldsymbol{K}$ is the covariance matrix of the observations.

Finally, one approach to the acquisition function for the choice of the next set of parameters is the Expected Improvement (EI) [Mockus et al., 1978] at a given point $\boldsymbol{\theta}$ defined as:

$$\mathrm{EI}(\boldsymbol{\lambda}) = \mathbb{E}_f[\max\{f(\boldsymbol{\lambda}) - f^*, 0\}]$$

where f^* is a target value, usually the best past observation/best posterior mean at past query point. f^* can be thought of as an aspiration value. EI attempts to do better than f^*, but instead of greedily exploiting, it also uses the estimates of uncertainty derived by the probabilistic model over f to explore the space $\boldsymbol{\Lambda}$.

3.10.4 Evolutionary Algorithm

Evolutionary algorithms have recently been successfully applied to hyperparameter optimization, architecture search, and hyperparameter optimization for neural networks [Real et al., 2017, Loshchilov and Hutter, 2016, Hansen et al., 2019]. As the name suggests, evolutionary algorithms imitate evolution; an initial population of sets of parameters/solutions also known as individuals is created. At each step, the fittest individuals are kept and reproduced. Crossing and mutation processes produce descendants, likely fitter than their parents. This process is performed iteratively for a given number of times until an optimal solution is found:

$$\text{Mutation} \rightarrow \text{Crossing} \rightarrow \text{Reproduction} \rightarrow \text{Selection}$$

Evolutionary methods can deal with different types of problems; non-linear, stochastic or discontinuous types. It does not require the use of a gradient. To evaluate each

candidate, an objective function known as fitness function is defined. This process is highly customizable but such a function can be defined as:

$$\textit{fitness function} = -2(\textit{average reward}) + \textit{error}$$

where the *average reward* is the average reward obtained in the last k episodes and *error* is the mean square error (MSE) of the reward in the last k episodes.

4 Implementation

In this section, we focus on the implementation of Q-learning, Deep Q-Network, and Actor-Critic model. We first describe how the environment is created. Secondly, we mention the complexity associated with dealing with a larger number of states and perform a short sensitivity analysis by changing the number of origins/destinations, the grid size, or the number of parcels/passengers. Then, we thoroughly describe each algorithm and its own hyperparameters. Finally, we detail our approach to the hyperparameter tuning process for each algorithm.

4.1 Environment

The environment is a crucial part of Reinforcement Learning (RL). In the present case, we build upon a very famous RL library called OpenAI. We explain the modifications we perform on the original case, go into more details with regard to our own modified version, and mention meaningful parts of the scripts used.

4.1.1 OpenAI

To create the environment, we use the OpenAI Gym framework in Python. OpenAI is "*an AI research and deployment company whose mission is to ensure that artificial general intelligence benefits all of humanity by which we mean highly autonomous systems that outperform humans at most economically valuable work - benefits all of humanity.*" It offers famous ready-to-use environments such as CartPole, Robotics or Atari. Most of these are used as benchmarks to compare different classes of algorithms. The one we will build upon is called "Taxi-v3".

The Basic Case

The Taxi-v3 environment was initially presented in [Diettrich, 2000]. This environment is represented as a 5-by-5 grid world with one passenger and one taxi agent (Figure 8). The passenger can appear randomly following a uniform distribution at five different locations named R, B, G, Y, and the one in the taxi. The taxi is coloured in yellow when there is no passenger inside or green when the passenger is in the taxi. The bold-coloured letter represents the origin and the other letter in colour is the destination. Walls are represented by a vertical bar, therefore the taxi agent cannot cross them. A free road

between two adjacent cases is represented by a colon. There is no horizontal wall in the original configuration. The task of the agent is to pick up the passenger and carry him/her to the destination. This is an episodic task. At each step, the taxi agent must choose an action $A_t \in \mathcal{A}(s)$. In this basic setting, the action space is of size 6, meaning six different actions are possible: South, North, East, West, Pickup passenger, and Drop off passenger. The state space is

$$5 \text{ rows} \times 5 \text{ columns} \times 5 \text{ possible locations} \times 4 \text{ possible destinations} = 500 \text{ states}$$

Each successful drop-off yields a $+20$ reward and each step costs -1. The goal of the taxi agent is to get the highest expected discounted reward.

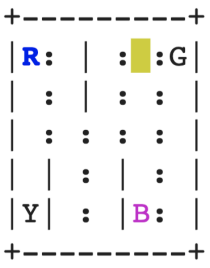


Figure 8: Taxi-v3 grid with an empty taxi in yellow, passenger in R and destination in B, an action space of 6, and a state space of 500.

The Modified Case

We modify the basic Taxi-v3 environment by following this set of instructions and create a new package called Delivery-v0 whose directory tree is as follows:

```
tpfood_gym/
  README.md
  agents
  videos
  setup.py
  gym_tpfood/
    __init__.py
    envs/
      __init__.py
      delivery_env.py
```

We use an environment with two parcels to be delivered from a given origin to a given destination. The environment is defined in the *delivery_ env.py* file. OpenAI uses Object-Oriented Programming (OOP) to build the environment which we follow. Every OpenAI

Gym environment, either original or customized is defined as a class with five different functions: a constructor *__init__*, a *step* function, a *reset* function, a *render* function, and a *close* function. These generic functions may be complemented by environment-specific functions. Let us note that class inheritance plays a role in the construction of a customized environment from the OpenAI gym library. To build a discrete environment from this library, one has to call the *Env* class into a *Discrete* class, itself called into our customized class *Delivery*.

In the case of the Delivery-v0 environment we have the following functions:

- *__init__*
- *step*
- *reset*
- *render*
- *close*
- *seed*
- *encode*
- *decode*

We begin by modifying the grid map, defined as an array. We want to add horizontal walls (Figure 9). To do so, we add $n-1$ vertical rows. These are "invisible" rows, meaning that their sole purpose is to define if there is a wall (-) or not (:). They are not visible to the agent. We then move on to customizing the different functions presented above.

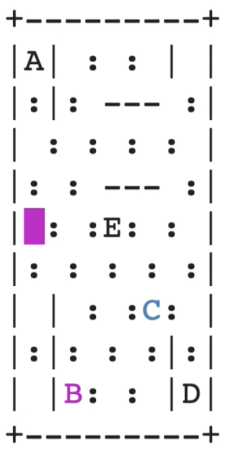


Figure 9: Delivery-v0 grid with a delivery man with parcel 2 on board, hence the magenta colour. Parcel 1 had to be delivered in C and parcel 2 must be delivered in B. The action space is of size 8 and the state space is of size 22500.

__init__ **:** this function is the constructor where we instantiate objects which will be useful in other functions. In the present case, the constructor is a default one, meaning

that it only accepts one argument; *self*. Inside, we define the grid map as an array, the number of rows and columns (integers), an array with the origins and destinations as well as the maximum number of rows, and the maximum number of columns. The array for origins and destinations, *locs*, is the same and contains coordinates on the grid. These are sampled randomly following a discrete uniform distribution. As Python is a 0-indexed programming language, we must subtract every value with 1. The number of states is calculated by using the following formula:

$$\begin{aligned}&\#\text{rows} \times \#\text{columns} \times (\#\text{possible locations} + 1)_{\text{parcel 1}} \times \#\text{possible destinations}_{\text{parcel 1}} \\ &\times (\#\text{possible locations} + 1)_{\text{parcel 2}} \times \#\text{possible destinations}_{\text{parcel 2}} = \#\text{states}\end{aligned}$$

We define the initial state distribution, meaning that, for every state and every action associated with this state, the following list exists {probability, next state, reward, done}. We initialize every value to 0. The way the environment is defined requires to enumerate all the possible states. A state is defined by six different numerical features:

- **taxi_row**: the row of the taxi/delivery man
- **taxi_col**: the column of the taxi/delivery man
- **pass_loc_1**: the location of passenger/parcel 1 in the locs array
- **pass_loc_2**: the location of passenger/parcel 2 in the locs array
- **dest_idx_1**: the destination of passenger/parcel 1 in the locs array
- **dest_idx_2**: the destination of passenger/parcel 2 in the locs array

To build the state space, we loop over each of these features and enumerate all the states. After that, we define the actions. There are eight possible actions for the agent; four for the delivery man moves and four related to the parcels.

* **Action 0**: move south
* **Action 1**: move north
* **Action 2**: move east
* **Action 3**: move west
* **Action 4**: pickup parcel 1
* **Action 5**: drop off parcel 1
* **Action 6**: pickup parcel 2
* **Action 7**: drop off parcel 2

Each of the first four actions consists in determining if either a horizontal or vertical wall makes a given action impossible and if this is not the case, the action changes the position of the agent. Pickup and dropoff actions are different. Each pickup action checks if the delivery man is at the parcel's location, if the parcel is not already at the destination or not in the delivery truck. If every condition is met then the parcel is marked as being carried by the delivery man. The dropoff action is similar; it checks if the passenger is in the taxi and if the taxi is at the passenger's destination. If the conditions are fulfilled, then the passenger's location equals his destination. The case of pickups or dropoffs at wrong locations are also considered in the *if* statements.

By default, each movement on the grid costs -1 to the agent. Picking the parcel up at the right location yields no reward but trying to pick up a parcel at a wrong location penalizes the agent by -10. Dropping off the passenger at the right place gives a reward of 20. If a dropoff occurs at a wrong location, the agent receives a penalty of -10.

A final condition in the constructor is used to check when an episode ends. If both parcels have been delivered at their respective destinations, then an episode is considered as over. This is materialized by a variable $done = True$. Finally, a new state is encoded (see *encode* function) and a transition matrix P is created to determine which state S_{t+1} is subsequent to a given state S_t and a given action A_t, which reward is associated with action A_t and if state S_t is terminal or not. Let us remark that transitions are deterministic; the agent has a probability of 100% to reach the subsequent state associated with the action it has just chosen (e.g., if the agent chooses west, it goes west with a 100% probability).

encode : this function, called in the *constructor*, uses 6 numerical features (taxi_row, taxi_col, pass_loc_1, pass_loc_2, dest_idx_1, dest_idx_2) to code for a state characterized by a unique combination of these different features. It performs the following operation:

$$\text{state } \# = \text{taxi\_row} \times \text{taxi\_col} \times \text{pass\_loc\_1} \times \text{pass\_loc\_2} \times \text{dest\_idx\_1} \times \text{dest\_idx\_2}$$

decode : this function, used in the *render* function, decomposes a given state into its 6 features (taxi_row, taxi_col, pass_loc_1, pass_loc_2, dest_idx_1, dest_idx_2).

render : this function provides a visual representation of the environment. It highlights the origins and destinations of each parcel given a specific set of conditions. If the parcel has not been picked up nor delivered, both points are highlighted. Once the order has been cleared, the destinations are highlighted in their respective colours; cyan for parcel 1 and magenta for parcel 2. The parcels' locations are both bold and highlighted.

step: this function defines the transition between states using the transition matrix P and returns a tuple made of next state, reward, done, and probability. As said above, $done = True$ if the state is terminal and probability is always equal to 1.

seed: the *seed* function returns a seed provided by the *seed()* method. It is used to initialize a random number generator.

reset: the *reset* function is used to initialize a new episode randomly. It uniformly samples a new state to begin an episode.

close: this function closes the environment freeing up all the physics' state resources and requiring to create the environment again.

4.2 Agents

In this section, we present three different agents; a value-based tabular method (Q-learning), a value-based approximate method (DQN), and a policy-based method (Actor-Critic Model).

4.2.1 Q-learning

After importing the *gym* library as well as our custom library *gym_tpfood*, we set a seed = 42 to be able to replicate our results. We then import the environment and define a function *Q_learning_train* which takes five arguments:

- **env**: the environment
- $\boldsymbol{\alpha}$: the step-size/learning rate parameter determines to what extent new information overrides old information. If $\alpha = 0$, the algorithm only exploits previously acquired knowledge. If $\alpha = 1$, only the most recent information is considered.
- $\boldsymbol{\gamma}$: the discount parameter which determines future rewards. In the limiting case where $\gamma = 0$, future rewards do not matter, the agent is myopic and only considers

current rewards. If $\gamma = 1$, the sum of rewards may become infinite if no terminal state is reached.

- $\boldsymbol{\varepsilon}$: the probability of choosing a random action
- **episode**: the number of episodes to train the agent on the environment.

Inside the function, we define three plotting metric arrays *all_epochs, all_penalties, rewards* to count the number of steps, the number of penalties, and the rewards per episode respectively. We initialize a Q-table matrix to 0. Its dimensions equal that of the observation space (number of states) by the number of actions. We loop over the number of episodes. At each episode, we randomly reset the environment to a new state, generate an array *episode_rewards* as well as three scalar values, *epochs, penalties*, and *rewards*. We set *done* = *False* so that we do not start the episode in a terminal state.

We instantiate a *while* loop specifying that as long as the agent does not reach a terminal state it will do the following; in state S_t take a random action A_t out of the action space with a probability of ε, or choose the action whose action-value is the highest with probability $1 - \varepsilon$. Once this is done, the next state S_{t+1} associated with the highest action-value, the reward, and the information whether this is a terminal state or not are extracted. After that, the update of the Q-table occurs. The old action-value associated with (S_t, A_t) is replaced by the new action-value computed following Equation (1). If a penalty occurs, the scalar value *penalties* is increased by 1. The reward received in this iteration is appended to *episode_rewards*. If a terminal state is reached, this changes *done* = *True* and ends the *while* loop. To prevent very long (or infinite) runs, we choose a limit of 1000 steps for a given episode. If this limit is reached, the agent is terminated and the episode is considered as over, even though it has not been solved. Finally, we compute the sum of the rewards received at each step of a given episode and append it to the array *rewards*. Finally, for every 1000 episodes, we print the number of episodes on which the agent has been trained so far.

To extract results, we use two different functions, *Q_learning_train* and *Q_learning_test*. The sole difference between those two functions is that *Q_learning_test* uses a Q-table and an agent which have already been trained. Therefore, the agent only considers the highest action-value in each state and has zero chance of choosing a random action. This makes sense given that the only purpose of choosing ε is to ensure enough exploration.

4.2.2 Complexity Challenge

One of the main challenges for a tabular method happens when the number of states is too large. A larger state space makes the Q-table larger and has a large impact on the computation time.

As we would like to apply reinforcement learning techniques to a larger grid, we cannot neglect this issue. To be aware of this, we conduct a small sensitivity analysis by changing several features defining a state notably:

- the number of origins / destinations
- the number of parcels/passengers to be delivered
- the grid size

The results of the sensitivity analysis can be found in Subsection 8.4.

The first plot simply details how many states would be created if we used the original case and gradually increased the number of destinations. The second graph is more meaningful as this is the present case. We clearly see that if we simply added another origin/destination, the number of states would double to 44,100. The third plot is the most interesting one. What adds complexity to our problem is not the number of origins but the number of parcels/passengers. If we added one more parcel, the number of states would skyrocket to 675,000. Therefore, the number of states grows exponentially. The last graph indicates how the grid size would affect the state space. Its effect is somehow limited as adding four more rows and four more columns would result in 72,900 states instead of the current 22,500.

All these elements tell us that we need to use approximate methods to solve our problem. This is what we detail in further sections.

4.2.3 Stable Baselines

Agent implementations can be manual but several libraries offer the possibility to apply ready-to-use agents with customizable parameters. One of them is Stable Baselines, a fork of OpenAI Baselines, the standard library for Reinforcement Learning environments and algorithms. Stable Baselines offers a major structural refactoring compared to OpenAI

Baselines; it has a more unified structure of algorithms, more documentation of function, and additional algorithms.

4.2.4 Deep Q-Network (DQN)

To implement a Deep Q-Network we call the *Stable Baselines (SB)* library as well as standard libraries such as *Numpy*, *Pandas* and *Matplotlib*. Just like in the Q-learning case, we call the *gym* and *gym_tpfood* libraries. With SB, we can use several aforementioned extensions, notably Double-DQN, Dueling-DQN or DQN with Experience Replay implemented following the respective original papers in which they were presented. We create the environment, set the seed to 42, and define a directory where the model will be saved. The DQN method has 25 arguments but we will only use nine of them for simplicity reasons:

- **the policy**: the architecture of the neural network; whether this will be a Multilayer perceptron (MLP) or a Convolutional Neural Network (CNN) with or without layer normalisation
- **env**: the environment
- $\boldsymbol{\gamma}$: the discount parameter
- **exploration_fraction**: fraction of entire training period over which the exploration rate is annealed
- **learning rate $\boldsymbol{\eta}$**: learning rate for Adam optimizer
- **buffer size**: size of the replay buffer acting as a "sliding window" to update the target
- **verbose**: the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **double Q**: whether to enable Double Q-learning or not
- **prioritized replay**: whether prioritized replay buffer will be used or not

We define a callback function *CheckTraining* which will be useful to monitor the training part. Stable Baselines offers several built-in callback functions, however we define a custom one. *CheckTraining* inherits from *BaseCallback*, a function common to every callback function. *CheckTraining* requires three arguments; the frequency with which

it is used, the directory from which it can retrieve the results of the training, and the verbosity level. Each time it is called, it will print the average reward of the last 100 episodes. Finally, we use the method *learn* on the model we define above to start the training.

The DQN will therefore select an action a_t with probability ε or choose the action with the highest action-value based on the Q-function it approximates. It then observes the reward it receives and stores the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer. At every update, a batch of transitions is sampled to update the DQN. Another network, called the target network, is used to bring stability to the training and is updated at each k steps.

4.2.5 Actor-Critic Model

We use the library PyTorch to implement an Actor-Critic model (AC). PyTorch is an open-source deep learning library developed by the Facebook AI research group, based on Torch. The AC model is specifically built for the Delivery-v0 environment. While Stable Baselines does provide an Actor-Critic implementation with variants (SAC, ACER, ACKTR,...), we need to tailor our model to the way the environment is coded. We call standard libraries as well as the *gym* and *gym_tpfood* library. We define the following functions:

- class AC_model
  - *__init__*
  - *forward*
  - *select_action*
  - *update_weights*
- *unpack_arch*
- *plot_rewards*
- *encode_states*
- *decode_positions*

***class AC_model*:**

- ***__init__*** **:** In the constructor, we declare the architecture of a shared neural network by setting the number of hidden units as well as two heads in the final layer; one for the actor with as many nodes as actions (action head with eight nodes) and one for the critic (value head) with one node. We create two arrays to

save the actions taken and the rewards received.

- ***forward* :** In the *forward* function, we define how the model is going to run from input to output. ReLU activation functions are used throughout the network. The *forward* function returns a softmax distribution in action scores and state values as a scalar.

- ***select_ action* :** This function uses one-hot encoding for the number of features and turns this array into a tensor. It then propagates forward through the network, creates a categorical distribution characterized by probs, a value returned by the function *forward*. An action is then sampled and appended to the *save_ actions* array.

- ***update_ weights* :** This function applies optimization step to the network based on a trajectory where the loss is the sum of minus the expected return and the squared TD[5] error for the value function V. The rewards are normalized to stabilize the learning process. Then the losses for the *policy_ losses* and *value_ losses* are appended to their respective arrays. A Huber Loss function[6] is used for the critic as to be less sensitive to outliers. We set the gradients to zero and apply backpropagation.

***unpack_ arch*:** this function extracts the three features of the neural network; the input size, the number of hidden units, and the number of actions.

***plot_ rewards* :** *plot_ rewards* creates a plot with the reward per episode and the average reward over the last 10 episodes.

We define the environment, the number of hidden units, the discount factor γ. We define the architecture and pass this object to the class model *AC_ model*. We create the optimizer Adam and set a learning rate η.

We set a counter for the number of episodes passed. We create four arrays; one for the episodic reward, one for the accumulated episodic reward, one for the averaged episodic reward, and one for the standard deviation of the episodic reward. We set the best episodic reward to $-\infty$ to start with. We then run a *for* loop for a given number of episodes where

[5] Temporal Difference Learning

[6] $L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for} |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$

the algorithm selects an action with the *select_action*. We retrieve the transition given by the environment; the next state, the reward, and if the state is a terminal state or not. Then we document the reward and the accumulated episodic reward. When we reach a terminal state ($done == True$), the counter of episodes is updated, the episodic reward is appended to the *episodic_reward* array. The mean episodic reward and the standard deviation are computed over 10 episodes. If the best average episodic reward is found, then *best_avg_episodic_reward* is updated. A log is printed every n steps. We reset the environment for the next update and update the weights of the shared neural network.

4.3 Hyperparameter Tuning

The environment has context-specific hyperparameters (generic and arbitrary ones). For example, this can be the training duration either in time or number of steps or episodes. A long training duration allows the agent to interact with his environment for a longer period of time but this extends the entire RL learning process. Short training sessions shorten the entire RL learning process, but can cause the agent to know the environment improperly and thus not be able to exploit the full learning potential. Our three agents all have several parameters. It is thus necessary to find good values of these to have the best possible agent. However, hyperparameter tuning is a tedious task and a tradeoff between quality and speed must be made. As we outlined in Section 3.10, several approaches exist. To do so, we use Optuna.

4.3.1 Optuna

Optuna is an open-source automatic hyperparameter optimization software framework, particularly designed for machine learning. It offers great flexibility, can search large spaces and offers trial parallelization. Optuna is capable of dynamically constructing the search space, it accommodates with a wide variety of models both small and large. It formulates the hyperparameter optimization as a process of minimizing or maximizing an objective function which takes a set of hyperparameters as input and returns a score. Optuna gradually builds the objective function through the interaction with a trial object (evaluation of *object* function). The framework combines two types of sampling strategies for hyperparameters, which are called relative sampling and independent sampling. The relative sampling determines the values of multiple parameters simultaneously so that

sampling algorithms can use the relationship between parameters (e.g., correlation). The independent sampling determines the value of a single parameter without considering any relationship between parameters. Target parameters of the independent sampling are the parameters not described in the relative search space. Finally, let us note the possibility of using built-in procedures as well as our own sampling procedure. Out of simplicity, we will use the most basic sampling procedure, *base sampler*. The default sampler in Optuna is a Tree-structured Parzen Estimator (TPE), a form of Bayesian Optimization. The Tree-structured Parzen Estimator (TPE) is a sequential and independent sampling model-based optimization (SMBO) approach. SMBO methods sequentially construct models to approximate the performance of hyperparameters based on historical measurements, and then subsequently choose new hyperparameters to test based on this model.

4.3.2 Q-learning

To find the best hyperparameters, we define an *objective* function. This function contains the same code as that of Subsection 4.2.1. However, α, γ, and ε are now sampled according to discrete uniform distributions. Because computations can take a lot of time, we decide to use at most five values per parameter and choose these in a sensible manner, meaning that we will not try values which will likely yield bad results (e.g., $\alpha = 0$, $\varepsilon = 1$,...).

We test three different values for the step-size α : 0.3, 0.6, and 0.9. The discount factor γ can be equal to 0.6, 0.7, 0.8, 0.9, and 1. The exploration parameter ε can either be 0.01, 0.05 or 0.09. Therefore, there are 45 possible combinations. We use 1,000,000 episodes per trial. The metric used to assess each trial is the mean reward over all the episodes multiplied by -1. As we want to find the maximal reward, we need to use the opposite of the reward because Optuna is built for minimization. ($\max f(x) = -\min f(-x)$).

We run 30 trials and then extract the best value for each hyperparameter. We save all 30 trials in a dataframe in a csv format and display them under Section 8.5.1.

4.3.3 Deep Q-Network (DQN)

We define two functions to find the best hyperparameters. The first function, called *optimize_ dqn*, samples the hyperparameters used for tuning. The second function, *optimize_ agent*, is the *objective* function. This function contains the same code as that

of Subsection 4.2.4. Due to the large computational requirements, we choose three hyperparameters to optimize namely; the discount rate γ, the learning rate η, and the fraction of steps where exploration occurs as *exploration_fraction*, which we call ρ. Again, we decide to use some prior knowledge to exclude values which would yield bad results, such as a very low gamma, a high learning rate or a very low exploration fraction. γ can take on three different values; 0.4, 0.6 and 0.8. η can float between 0.00005 and 0.005. Finally, the exploration fraction ρ can oscillate between 0.5 and 0.7.

Inside *optimize_ agent* we call *optimize_ dqn*, create the environment and call the method DQN from Stable Baselines to create the agent. We use a Multi-Layer Perceptron with Layer Normalization (LnMLP) with two layers and 64 nodes each. We implement double Q-learning as well as Prioritized Experience Replay (PER). The buffer is of size 50,000. We do so out of the necessity to keep a buffer large enough to avoid a high correlation between samples while avoiding a too high computational cost. To create a metric to compare the different runs, we compute the mean reward over 10 test episodes. We return the negative value of that value as we want to maximize our objective function.

Because each run requires a lot of time, we only do six runs and parallelize them over two cores to accelerate the process. We train each model for 2,000,000 episodes. We do so out of the necessity to get good results and thus make good decisions while keeping the computation time as low as possible. The results can be found in the Appendix under Section 8.5.2.

4.3.4 Actor-Critic Model

We use the Optuna library and define a function named *objective*. We search for the best discount rate γ, the best learning rate η, and we test two different numbers of hidden units. γ can take on two values; 0.7 and 0.8. The learning rate η is sampled from a uniform distribution $\mathcal{U}(0.0005, 0.005)$. Finally, the hidden units can either be equal to 32 or 64. Again, we decide to choose sensible bounds for our parameters to speed up the computation time.

We use the same code as the one explained in Subsection 4.2.5 and run each trial for 50,000 episodes. This number of episodes is relatively small but as computation time is long, it should already give us a clear enough picture of where the training is going. As

each model can take more than one day to train, we choose to run five trials. The results can be found in the Appendix under Section 8.5.3.

5 Results

5.1 Q-learning

We present the results of the Q-learning algorithm for both the training (Figures 10 and 11) and the testing parts (Figure 12). The training part enables us to see if and how fast the agent converges. We also get a visual representation of the performance of three different agents; the one associated with the best combination of hyperparameters, the one associated with the worst combination of hyperparameters, and a third random combination[7] for illustrative purposes. The first finding easily noticeable is the rapidity of convergence during training (Figure 10). Every agent converges in less than 30,000 episodes. When we zoom on the first 60,000 episodes (Figure 11), we distinguish that the best agent ($\alpha = 0.9, \gamma = 0.8, \epsilon = 0.01$) breaks even in less than 5,000 episodes and reaches an average score per episode of 20.67. The worst combination in our trials performs almost half as well, breaking even in less than 10,000 episodes and reaching an average reward per episode of slightly more than 12.39. Finally, the random combination is the slowest one, reaching 0 only after 15,000 episodes and getting an average episode reward of less than 10.52. Let us note that both the second and third agent need respectively twice and three times as much time as the first one to reach 0, and are half as good as the first agent.

When we compare all three agents in the testing phase[8], results are even clearer. Agent 1, in blue, has a low variance and oscillates around 21.70. Although agent 2 has almost the same performance as agent 1 on average with 21.05 per episode, it has a larger variance and occasionally experiences dips in episode rewards. Finally, agent 3, which is made of a random hyperparameter combination, has the worst performance having an average performance of -6.39 per episode and experiences an enormous variance. Some episodes are finished with a positive reward while others are negative. For a real-world application, this agent would be directly discarded. As a final note, let us remember that each passenger yields a reward of 20 to the agent. This means that the maximum it can get is 40. As the best agent has an average reward per episode of 21, we can deduce that the average number of steps per episode (if no penalty occurs) is 19, which is fairly good.

[7] Taken from the range of hyperparameters we chose for hyperparameter tuning, this combination was not tested during the trials.

[8] There is no more exploration ε.

5.2 Deep Q-Network

We present the results of the Deep Q-Network for the training (Figures 13, 14, 15, and 16) and the testing parts (Figure 17). We train and plot three different agents; the best agent in purple ($\gamma = 0.8$, $\eta = 0.0019, \rho = 0.671$)[9] resulting from the hyperparameter optimization, the third-best in orange ($\gamma = 0.6$, $\eta = 0.0003, \rho = 0.669$), and the worst one in light blue ($\gamma = 0.4$, $\eta = 0.0017, \rho = 0.502$) for 3,000,000 timesteps. To train our model, we do not specify the number of episodes but the number of timesteps, meaning that the most efficient agent will be the one which goes the furthest. From Subsection 8.5.2, we can state that the exploration fraction seems positively correlated with the quality of the reward. So does a higher γ. In Figure 13, we directly notice how difficult it is for the worst agent to reach even 20,000 episodes. Both the best and third-best agent break even and converge. Both models trend above 0 already after 20,000 episodes (Figure 14). A surprising finding is the rapidity of the second model to pass the 0 threshold. It is even faster than the best model. However, its variance is also higher and dips are clearly noticeable for both models. Looking at the very end of the training process (Figure 15), the third-best model only reaches 72,000 episodes while our best model can reach 97,500 episodes. The same findings can be remarked when we plot for the number of timesteps (Figure 16). When we test the best model on 100 episodes (Figure 17), we get an average reward per episode of 22.46 (red line). While the average is better than the results obtained with Q-learning, the variance is greater. Some episodes may get a reward as low as 5 while others have a reward over 30.

5.3 Actor-Critic Model

We present the results of the Actor-Critic model for the training (Figure 18) and the testing part (Figure 19 and 20). We train and plot one agent, the best one from the hyperparameter tuning part. This way, we can save computational capabilities as one agent is trained in 10 to 12 hours. The neural network associated with the agent is made of 32 units/nodes. The discount rate γ equals 0.7 and the learning rate η is set to 0.003 with an Adam optimizer. We train this agent for 100,000 episodes. The training plot displays the reward per episode as well as a moving average over 10 episodes. After

[9]Note: we define ρ as the exploration fraction of the number of episodes where ε, the probability of choosing a random action, is annealed.

20,000 episodes, the agent comes closer to 0. Interestingly though, sudden dips cannot be excluded even after 20,000 where a huge dip of the reward of an episode reaches as low as $-200,000$. We test our agent on 100 and 1000 episodes respectively. We notice the same pattern as during the training phase. While the mean episodic reward equals 18.38 and 14.03, sudden drops are clearly noticeable, reaching as low as -350 over a single episode. The mean reward is lower than that of DQN and Q-learning and the variance is greater.

5.4 Comparison Of The Three Algorithms

As mentioned in Section 5.2, DQN and AC exhibit greater variance than Q-learning. This may be a problem because, although DQN is better on average, a greater variance leads to instability. Several explanations can be discussed. First, let us focus on DQN. The first explanation concerns the hyperparameter optimization process. It may be the case that, as we narrowed the possible values of hyperparameters to a relatively small range for computational reasons, this set of parameters is not the best. More iterations may have helped the Tree Parzen Estimator (TPE) to correctly identify better hyperparameters. Secondly, other parameters which we did not include in the optimization process may play a role. The replay memory size, a sliding window for prioritized experience replay, was maybe too small and states drawn for the minibatch updates were too heavily correlated. Finally, the target network which keeps a copy of a former version of the network training may not have been updated enough and hence biased results with beliefs of an outdated version of the neural network. Finally, bootstrapping, meaning updating estimates based on other estimates leads to greater variance. With regard to the Actor-Critic model, several explanations may be purported. Again as we narrowed the possible values of hyperparameters to a relative small range for computational reasons, this set of parameters is probably not the best. Secondly, the design of the neural network is probably not optimal as the instability suggests. It may have been better to include more hidden layers. Also, the baseline chosen for the AC model (mean reward over the last 10 episodes) may be improved. Thirdly, there may be a vanishing gradient problem which could be tackled with regularization or dropout layers. While the AC model is the worst in terms of reward it is also the most promising one for larger state space and thus should not be discarded. Finally, as the AC model chooses actions probabilistically, a small number of episodes may have impeded it to converge towards a more optimal distribution for action selection.

5.5 Identification Of Ride-Sharing Opportunities

While we have put much focus on the mean episodic reward, investigating whether ride-sharing is possible is also our goal. We are interested to know if the Q-learning and AC model would use a different strategy for ride-sharing. In fact, we aim to determine if the delivery man only performs one trip when the two parcels need to be picked up at and delivered to the same place. To this end, we filter the Q-table of the optimal agent (Section 8.7) of the Q-learning algorithm for states where

$$\text{Q-value}_{\text{ pickup passenger 1}} = \text{Q-value}_{\text{ pickup passenger 2}}$$
$$\text{Q-value}_{\text{ dropoff passenger 1}} = \text{Q-value}_{\text{ dropoff passenger 2}}$$

The rationale behind this condition is that, as Q-values reflect the expected return starting from s, taking action a and thereafter following policy τ, pickup actions and dropoff actions of the two parcels should have the same values.

We identify at least four such states (12, 17087, 19094, and 22125). These are displayed in Figures 21 and 22. Let us note that other situations where the two passengers/parcels could use ride-sharing but are not located at the same place and do not want to go to the same destination are also possible. These would be hard to identify as we could not recover them in the Q-table by filtering it because the action-values would have different values.

When we test the Q-learning agent starting from state 12, we clearly see that, although ride-sharing would be possible, the agent does not grasp this opportunity and drives each passenger after the other. We visually check the other cases and the same conclusion is reached. This could happen because the Q-learning algorithm only updates its estimates based on one step ahead (TD(0)). It may thus lack a long-term view in an episode. Secondly, only 4 cases out of 22,500 were identified as perfect cases for ride-sharing. The sample may be too small for the algorithm to learn appropriately how to handle such situations.

On the other hand, the Actor-Critic agent exhibits different behaviour. Each opportunity for ride-sharing is taken and ride-sharing is favoured. The episode starting with state 12 has 2 passengers/parcels going from point A to point C. Both do ride-sharing and are

correctly delivered. In the episode starting with state 19094, passengers/parcels 1 and 2 are located in B, parcel 2 needs to go from B to E while parcel 1 has to go from B to D. The algorithm correctly identifies an opportunity for ride-sharing and picks up both parcels at the same time even though they have two different destinations. The episode starting with state 22125 has two parcels located in D which need to go to A. While the ride-sharing opportunity is taken, the trajectory is not optimal as the agent does not drop off straight after parcel 1, goes south one step, and then comes back.[10] This is good news indeed. While the agent may not be optimal, it has developed a policy for such cases nonetheless.

The video of a Q-learning agent playing the episode starting with state 12 can be found here. The videos of episodes with an AC agent starting with states 12, 19094, and 22125 can be found here.

[10]Episode starting with state 17087 is not included as the agent only has to move two cases away to reach the end of it.

6 Limitations

As mentioned several times throughout this work, one of the main drawbacks of current Reinforcement Learning (RL) is the computational capabilities required to train agents. While tabular methods are more easily trained even on a laptop, more complex methods involving Deep Learning require many more hours if not days. Moreover, finding optimal hyperparameters is a tedious task. Secondly, Reinforcement Learning differs from other machine learning methods in several ways. The data used to train the agent is collected through interactions with the environment by the agent itself (compared to supervised learning where you have a fixed dataset for instance). This dependence can lead to a vicious circle: if the agent collects poor quality data (e.g., trajectories with no rewards), then it will not improve and continue to amass bad trajectories. This also affects the way a value estimator (using function approximation) can capture the true reward structure of the environment. Thirdly, one main difference between RL and supervised learning is that RL does not have an objective as clearly defined as that of supervised learning (SL). While metrics such as accuracy, cross-entropy or mean squared error are readily available in SL, the definition of a metric for a RL problem remains elusive and is highly context-dependent. Designing a reward function is not an easy task. While this food-delivery/ride-sharing problem maximal reward was capped slightly below 40 (two parcels delivered correctly), some tasks may have no ceiling and it is thus harder to determine how good an agent is. Benchmarks are thus needed and in this case, the *OpenAI gym* library is useful. Fourthly, other issues may arise such as the difficulty to escape a local optima, overfitting or the difficulty of reproducibility. Indeed, changing the seed may have an impact on the final results. Finally, the wide array of RL agents available requires a trial-and-error approach to find the best possible agent for a specific task. Generalization is not easily possible.

7 Discussion And Conclusion

After a thorough examination, the Actor-Critic agent remains the favourite agent for the food-delivery/ride-sharing task. Although it is disappointing in terms of overall performance, its ability to develop strategies to increase efficiency in delivery is very promising. Moreover, the instability problem associated with the actor could probably be solved with a better hyperparameter tuning process, a better design of the neural network, and more computational capabilities. Further extensions could be considered such as implementing a true Advantage Actor-Critic model (A2C) or an Asynchronous Advantage Actor-Critic (A3C) model which are the state-of-the-art in the Actor-Critic class. Both would bring more stability to the agent. The present case was only made of two passengers and five destinations but it does offer some limited insights on how interesting real-world applications could be.

Despite its drawbacks, RL remains an exciting subfield of Machine Learning with much prospect in the coming years, from Natural Language Processing to autonomous driving cars, supply chain or financial trading. New algorithms or new extensions of existing algorithms are developed every year and most of the papers used for the present work are fairly recent, mostly written five years ago or less. This is also the case for the libraries used for the tasks. OpenAI (2015), PyTorch (2016), Stable-Baselines (2018) or Optuna (2019) are all proof of the large and recent interest in ML and RL in the last years. Developments to address the RL drawbacks are also underway. One of them worth mentioning is the pruning of Deep Reinforcement Learning agents which could accelerate the training of DQN or AC algorithms. The present work has also been a personal opportunity to discover, learn, implement, and reflect on a new interesting topic as well as on a new programming language, Python. Despite learning about ML almost two years ago, long waiting times to train algorithms were unknown to me. This has forced me to approach them with patience, a more structured methodology to tackle problems and develop a "start small think larger" mindset.

8 Appendix

8.1 Q-learning

Algorithm 1 Q-learning (off-policy TD control) for estimating $\pi \approx \pi^*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of an episode:
        Choose A from S using policy derived from Q (e.g. ε-greedy)
        Take action A, observe R, S'
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
        $S \leftarrow S'$
    until S is terminal

8.2 DQN

Algorithm 2 Deep Q-learning with Experience Replay

Initialize network Q with random weights $\boldsymbol{w}$
Initialize target network $\hat{Q}$ with weights $\boldsymbol{w}^- = \boldsymbol{w}$
Initialize replay memory $\mathcal{D}$ to capacity C
Initialize the agent to interact with the environment

while *not converged* **do**

    Sampling phase
    ε set new epsilon with ε-decay
    Choose action a from state s using policy ε-greedy(Q)
    Agent takes action a, observe reward r, and next state s'
    ε set new epsilon with ε-decay
    Store transition $(s, a, r, s', done)$ in experience replay memory $\mathcal{D}$
        if enough experiences in $\mathcal{D}$ **then**

            Learning phase
            Sample a random minibatch of N transitions from D
                if $done_i$, **then**
                    $y_i = r_i$
                else
                    $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
                end
            end
            Calculate the loss $\mathcal{L} = 1/N \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
            Update Q using the SGD algorithm by minimizing $\mathcal{L}$
            Every T steps, copy weights $\boldsymbol{w}$ from Q to $\hat{Q}$ as $\boldsymbol{w}^-$
    end for
end for

8.3 Policy-Based Algorithms

8.3.1 REINFORCE

Algorithm 3 REINFORCE with baseline (episodic) for estimating $\pi_{\boldsymbol{\theta}} \approx \pi^*$

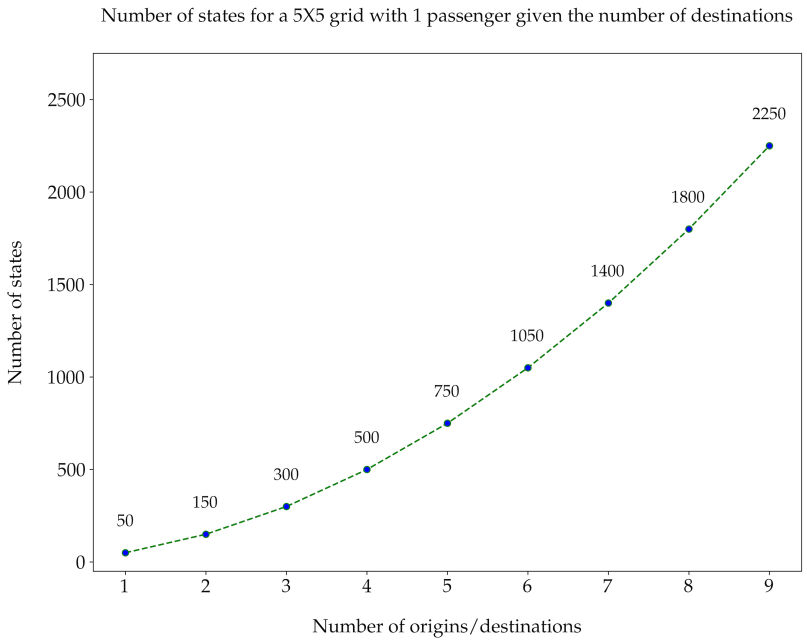
Input: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\boldsymbol{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ (e.g, to $\mathbf{0}$)
For each episode:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of an episode $t = 0, 1, \cdots, T-1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A|S_t, \boldsymbol{\theta})$

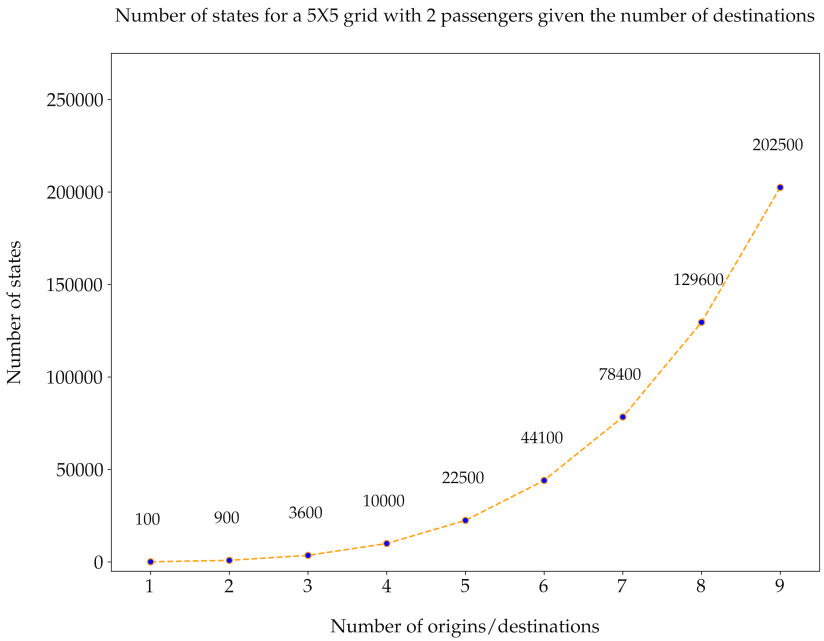
8.3.2 Actor-Critic Model

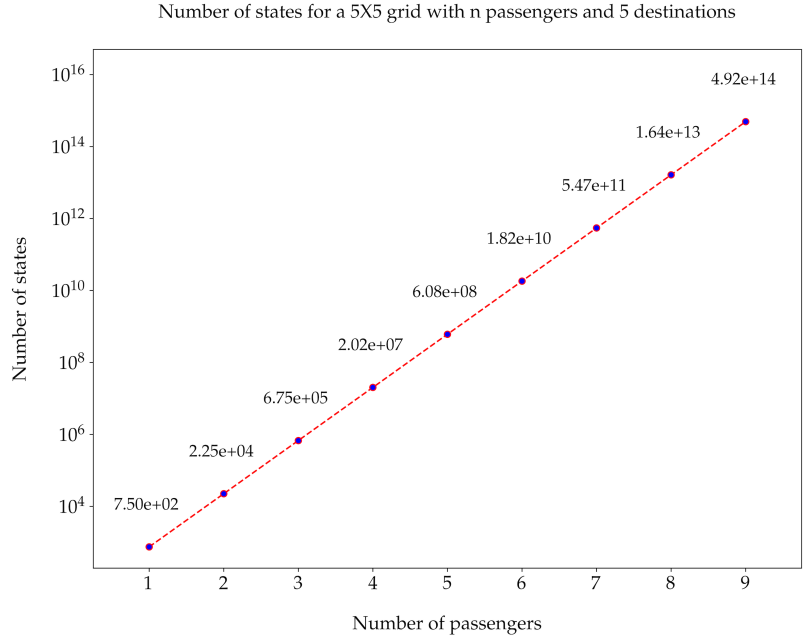
Algorithm 4 One-step Actor-Critic (episodic) for estimating $\pi_{\boldsymbol{\theta}} \approx \pi^*$

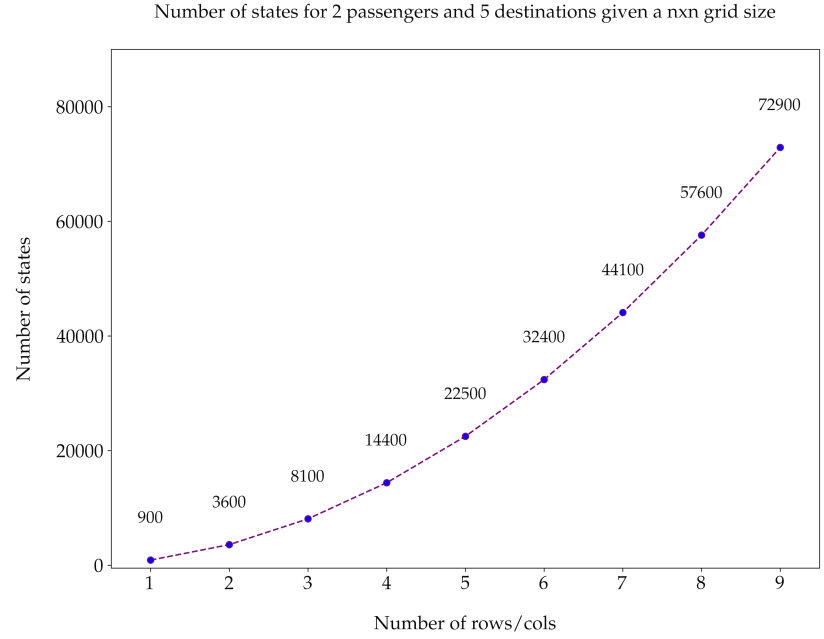
Input: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parametrization $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\boldsymbol{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$
For each episode:
    Initialize S, the first state of an episode
    $I \leftarrow 1$
    While S is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action A, observe S', R
        $\delta \leftarrow R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S', \boldsymbol{w})$ (if S' is terminal, then $\hat{v}(S', \boldsymbol{w}) \doteq 0$)
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S, \boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$

8.4 Complexity Issue









8.5 Hyperparameter Tuning

8.5.1 Q-learning Trials

Number	Value	Datetime start	Datetime complete	Duration	Alpha	Gamma	Epsilon	state
0	-16.19	28.09.20 22:50:37	28.09.20 22:57:34	0 days 00:06:57	0.9	1	0.05	COMPLETE
1	-15.836	28.09.20 22:57:34	28.09.20 23:04:35	0 days 00:07:01	0.6	0.9	0.05	COMPLETE
2	-18.861	28.09.20 23:04:35	28.09.20 23:11:47	0 days 00:07:11	0.3	0.9	0.01	COMPLETE
3	-18.772	28.09.20 23:11:47	28.09.20 23:19:25	0 days 00:07:37	0.3	1	0.01	COMPLETE
4	-15.625	28.09.20 23:19:25	28.09.20 23:26:45	0 days 00:07:20	0.6	0.6	0.05	COMPLETE
5	-15.178	28.09.20 23:26:45	28.09.20 23:35:19	0 days 00:08:34	0.3	0.7	0.05	COMPLETE
6	-12.389	28.09.20 23:35:19	28.09.20 23:43:45	0 days 00:08:25	0.9	1	0.09	COMPLETE
7	-15.153	28.09.20 23:43:45	28.09.20 23:51:30	0 days 00:07:44	0.3	0.9	0.05	COMPLETE
8	-11.492	28.09.20 23:51:30	29.09.20 00:00:05	0 days 00:08:34	0.6	0.6	0.09	COMPLETE
9	-19.343	29.09.20 00:00:05	29.09.20 00:06:56	0 days 00:06:51	0.6	0.9	0.01	COMPLETE
10	-19.574	29.09.20 00:06:56	29.09.20 00:14:28	0 days 00:07:32	0.9	0.8	0.01	COMPLETE
11	-19.562	29.09.20 00:14:28	29.09.20 00:21:18	0 days 00:06:49	0.9	0.8	0.01	COMPLETE
12	-19.568	29.09.20 00:21:18	29.09.20 00:27:53	0 days 00:06:34	0.9	0.8	0.01	COMPLETE
13	-19.509	29.09.20 00:27:53	29.09.20 00:35:27	0 days 00:07:33	0.9	0.8	0.01	COMPLETE
14	-19.463	29.09.20 00:35:27	29.09.20 00:42:48	0 days 00:07:21	0.9	0.7	0.01	COMPLETE

Number	Value	Datetime start	Datetime complete	Duration	Alpha	Gamma	Epsilon	state
15	-19.442	29.09.20 00:42:48	29.09.20 00:50:23	0 days 00:07:34	0.9	0.7	0.01	COMPLETE
16	-19.542	29.09.20 00:50:23	29.09.20 00:57:17	0 days 00:06:53	0.9	0.8	0.01	COMPLETE
17	-19.384	29.09.20 00:57:17	29.09.20 01:04:02	0 days 00:06:45	0.6	0.7	0.01	COMPLETE
18	-15.978	29.09.20 01:04:02	29.09.20 01:11:02	0 days 00:06:59	0.9	0.8	0.05	COMPLETE
19	-12.013	29.09.20 01:11:02	29.09.20 01:17:27	0 days 00:06:24	0.9	0.8	0.09	COMPLETE
20	-19.376	29.09.20 01:17:27	29.09.20 01:23:14	0 days 00:05:47	0.6	0.7	0.01	COMPLETE
21	-19.543	29.09.20 01:23:14	29.09.20 01:28:58	0 days 00:05:44	0.9	0.8	0.01	COMPLETE
22	-19.487	29.09.20 01:28:58	29.09.20 01:34:44	0 days 00:05:45	0.9	0.9	0.01	COMPLETE
23	-19.475	29.09.20 01:34:44	29.09.20 01:40:30	0 days 00:05:46	0.9	0.8	0.01	COMPLETE
24	-19.487	29.09.20 01:40:30	29.09.20 01:46:17	0 days 00:05:46	0.9	0.8	0.01	COMPLETE
25	-19.477	29.09.20 01:46:17	29.09.20 01:52:00	0 days 00:05:43	0.9	0.7	0.01	COMPLETE
26	-16.235	29.09.20 01:52:00	29.09.20 01:57:56	0 days 00:05:55	0.9	0.9	0.05	COMPLETE
27	-19.384	29.09.20 01:57:56	29.09.20 02:03:42	0 days 00:05:46	0.6	0.8	0.01	COMPLETE
28	-19.484	29.09.20 02:03:42	29.09.20 02:09:28	0 days 00:05:45	0.9	0.7	0.01	COMPLETE
29	-16.216	29.09.20 02:09:28	29.09.20 02:15:22	0 days 00:05:54	0.9	0.9	0.05	COMPLETE

8.5.2 DQN Trials

Number	Value	Datetime Start	Datetime Complete	Duration	Exploration fraction	Gamma	Learning rate	State
0	-22.4	14.10.20 15:30:03	15.10.20 10:51:51	0 days 19:21:47	0.664	0.8	0.0013	COMPLETE
1	1000	14.10.20 15:30:03	15.10.20 10:42:36	0 days 19:12:33	0.502	0.4	0.0017	COMPLETE
2	-26.2	15.10.20 10:42:36	16.10.20 06:12:14	0 days 19:29:37	0.671	0.8	0.0019	COMPLETE
3	896.9	15.10.20 10:51:51	16.10.20 06:22:24	0 days 19:30:33	0.513	0.6	0.0001	COMPLETE
4	179.6	16.10.20 06:12:14	17.10.20 02:20:10	0 days 20:07:56	0.669	0.6	0.0003	COMPLETE
5	384.4	16.10.20 06:22:24	17.10.20 02:27:50	0 days 20:05:26	0.533	0.6	0.0001	COMPLETE

8.5.3 Actor-Critic Trials

Number	Value	Datetime start	Datetime Complete	Duration	Gamma	Hidden units	Learning rate	State
0	1707.1	31.10.20 17:48:31	01.11.20 05:24:24	0 days 11:35:52	0.8	32	0.0028	COMPLETE
1	1821.6	01.11.20 05:24:24	01.11.20 14:21:02	0 days 08:56:38	0.8	32	0.0019	COMPLETE
2	1474.7	01.11.20 14:21:02	02.11.20 04:42:18	0 days 14:21:16	0.7	32	0.003	COMPLETE
3	2118.7	02.11.20 04:42:18	02.11.20 13:59:07	0 days 09:16:49	0.8	32	0.0021	COMPLETE
4	2310.5	02.11.20 13:59:07	03.11.20 08:06:29	0 days 18:07:21	0.8	32	0.0025	COMPLETE

8.6 Results

8.6.1 Q-learning

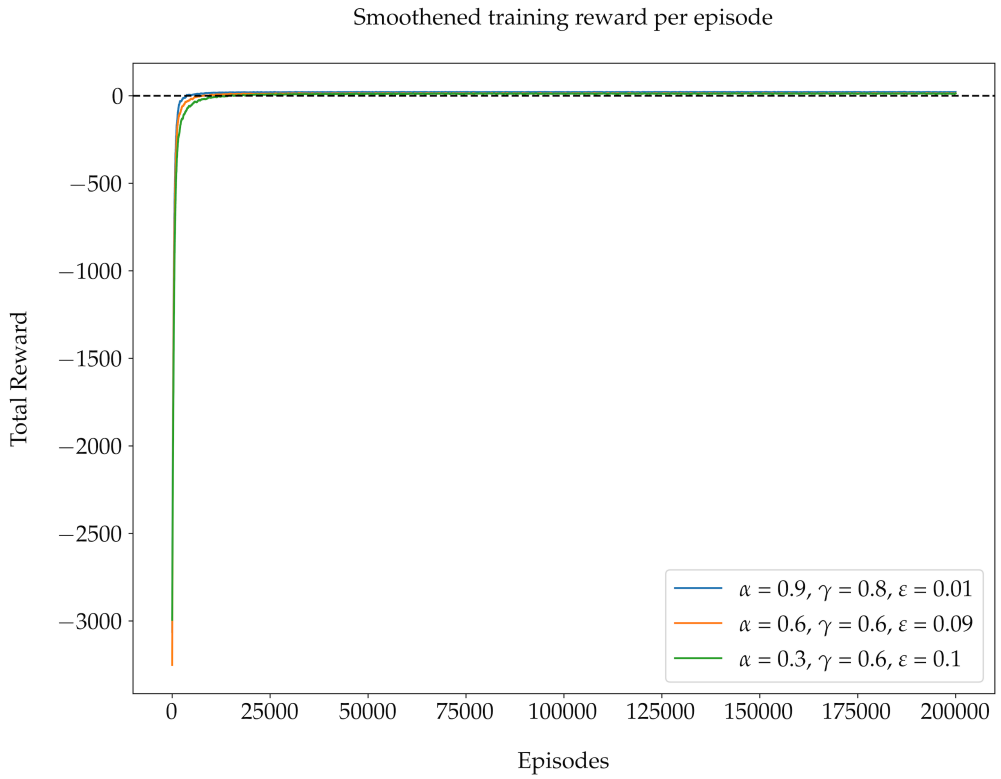


Figure 10: Smoothened training reward per episode up to the first 200,000 episodes

Figure 11: Smoothened training reward per episode up to the first 60,000 episodes

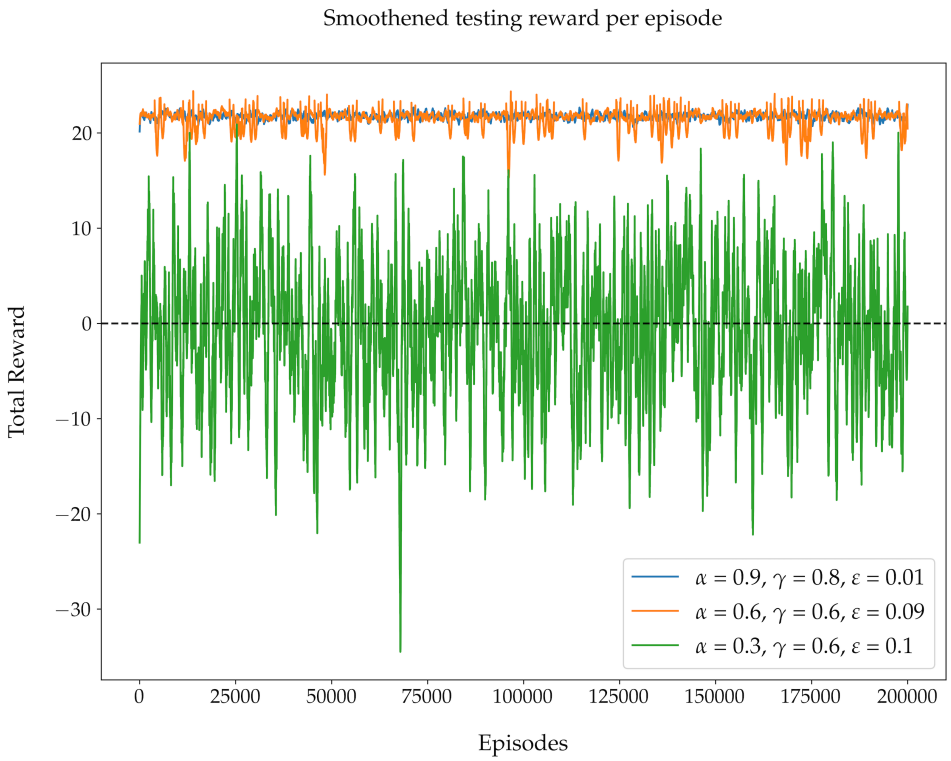


Figure 12: Smoothened testing reward per episode for all episodes

8.6.2 DQN

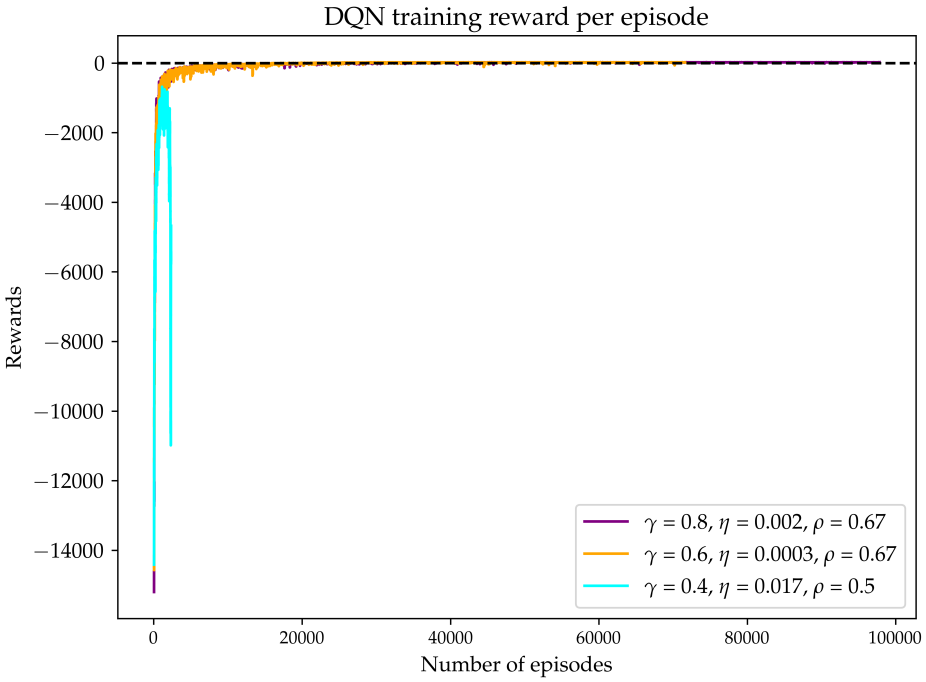


Figure 13: Smoothened training reward per episode up to the first 100,000 episodes

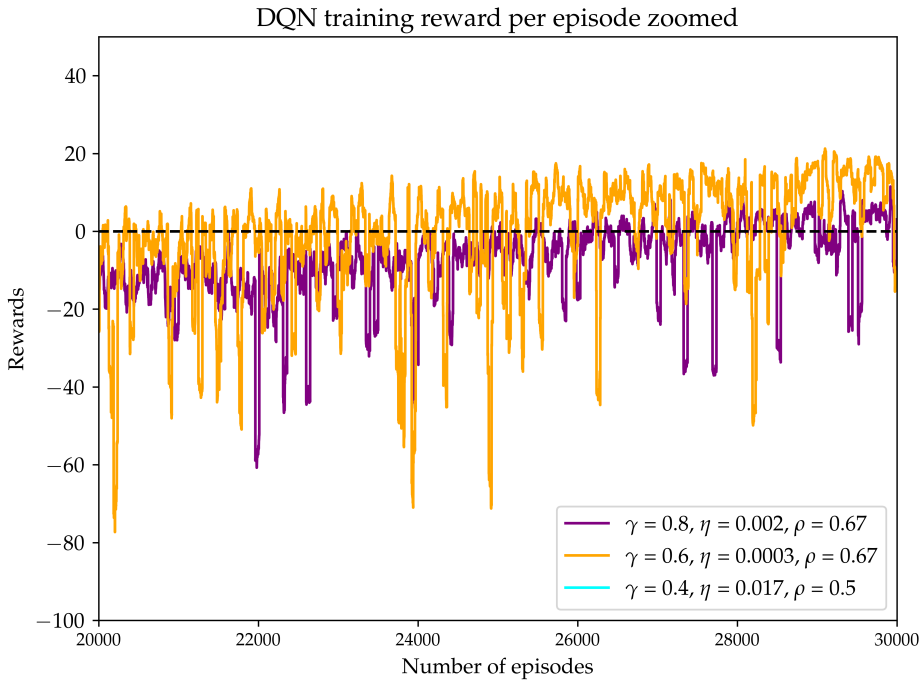


Figure 14: Zoomed smoothened training reward per episode between 20,000 and 30,000 episodes

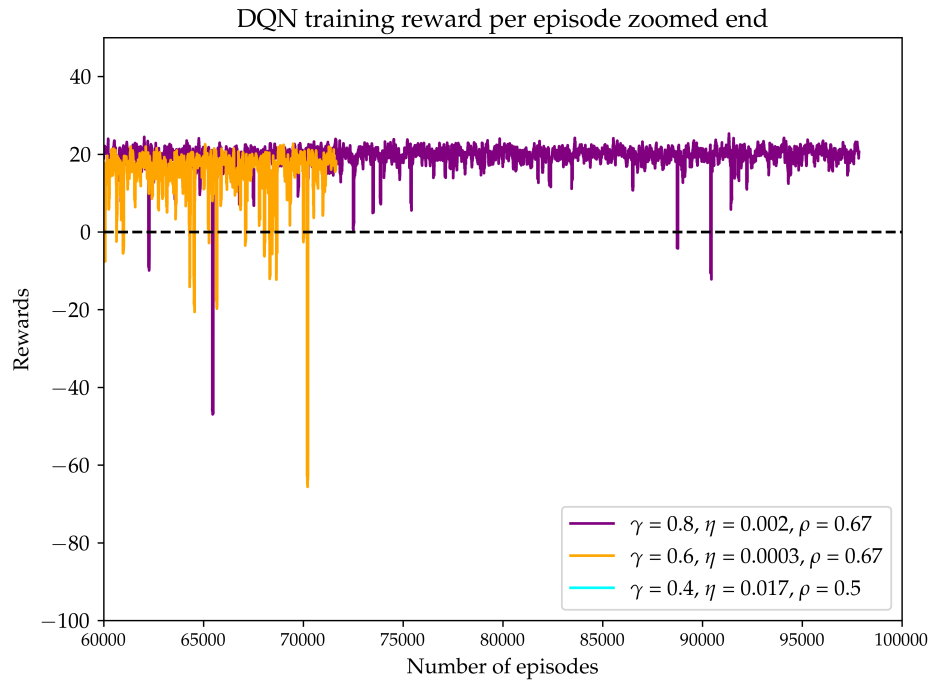


Figure 15: Zoomed smoothened training reward per episode up from 60,000 to 100,000 episodes

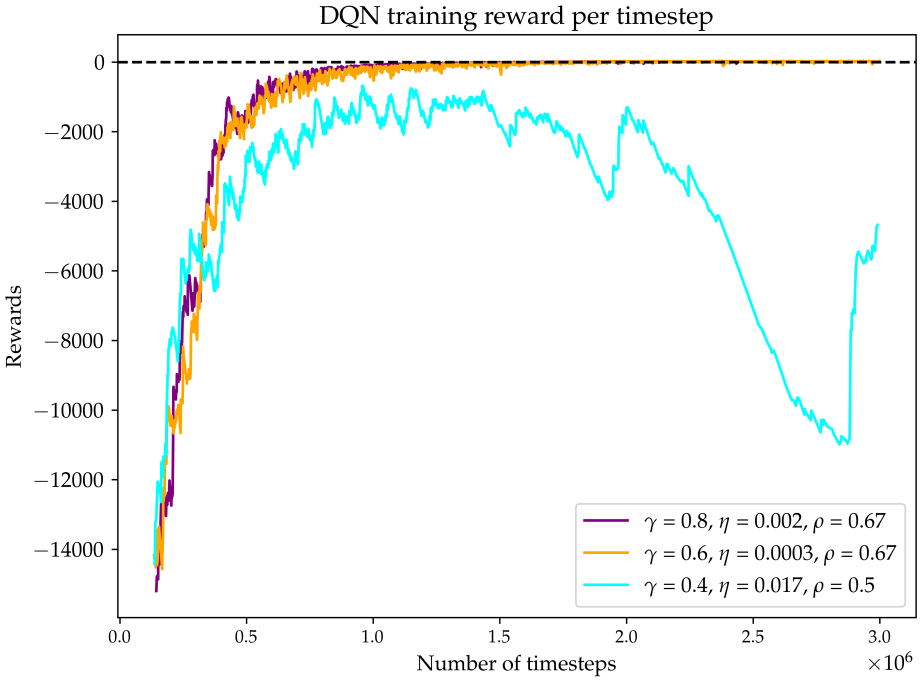


Figure 16: Smoothened training reward per timestep up to 3,000,000 timesteps

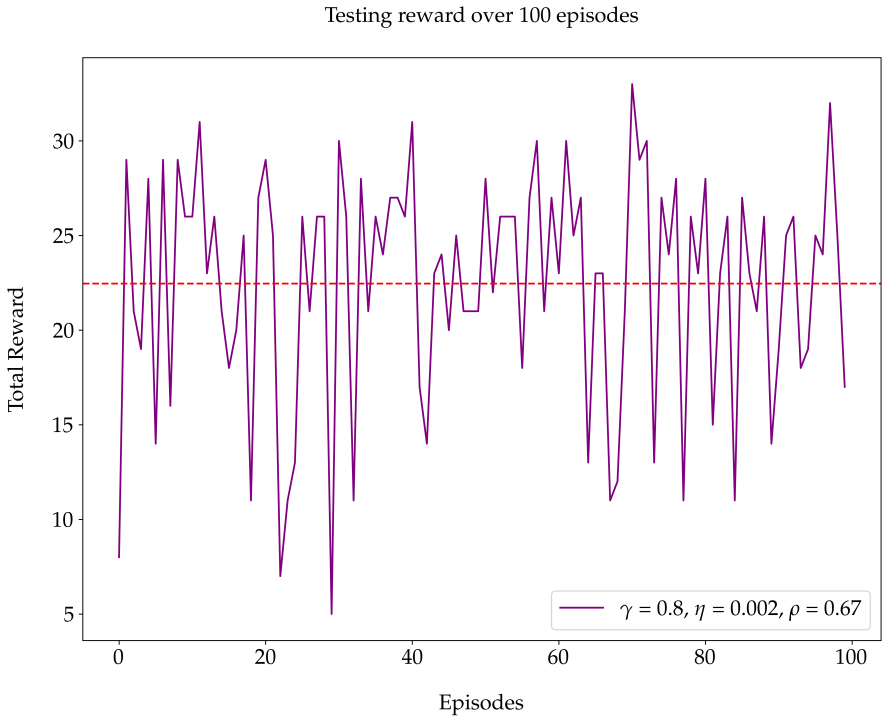


Figure 17: Testing reward for the best model for 100 episodes

8.6.3 Actor-Critic

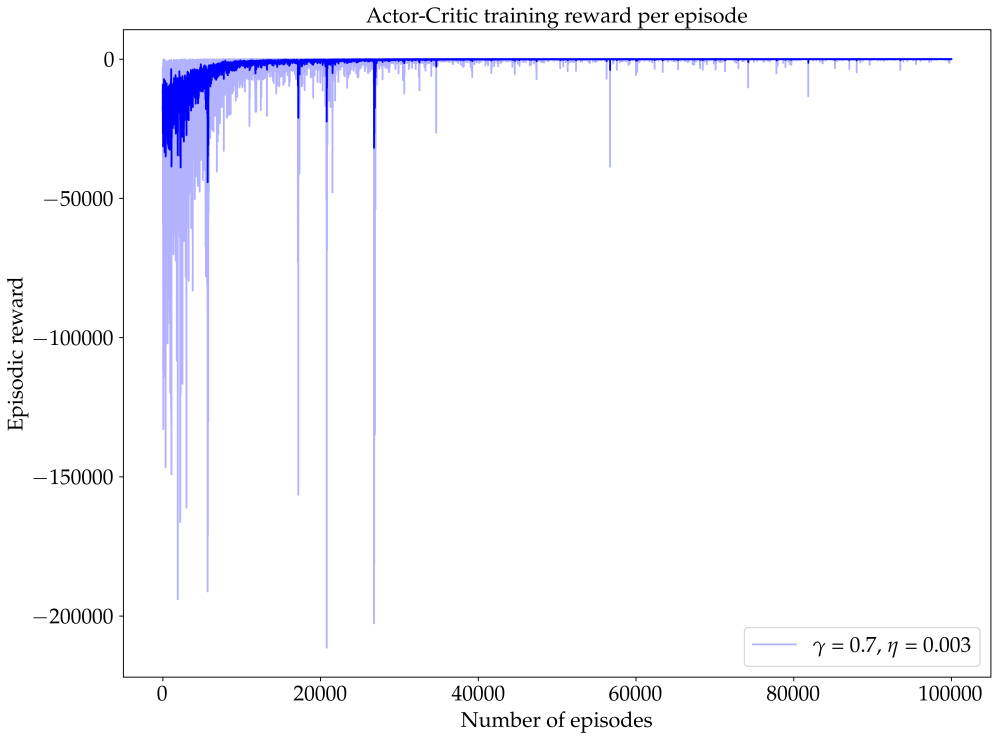


Figure 18: Smoothened training reward per timestep up to 100,000 episodes timesteps

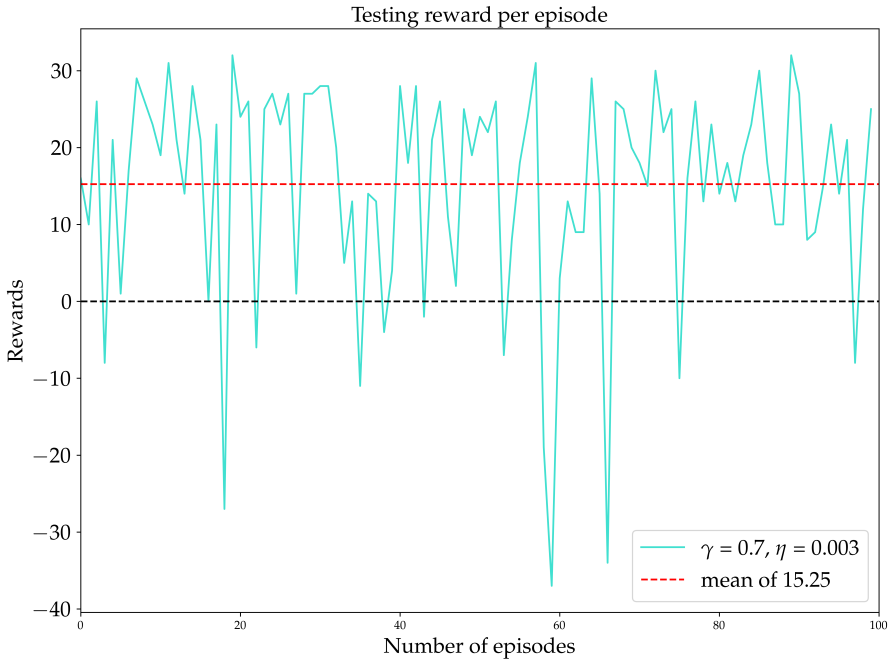


Figure 19: Testing reward for the best model for 100 episodes

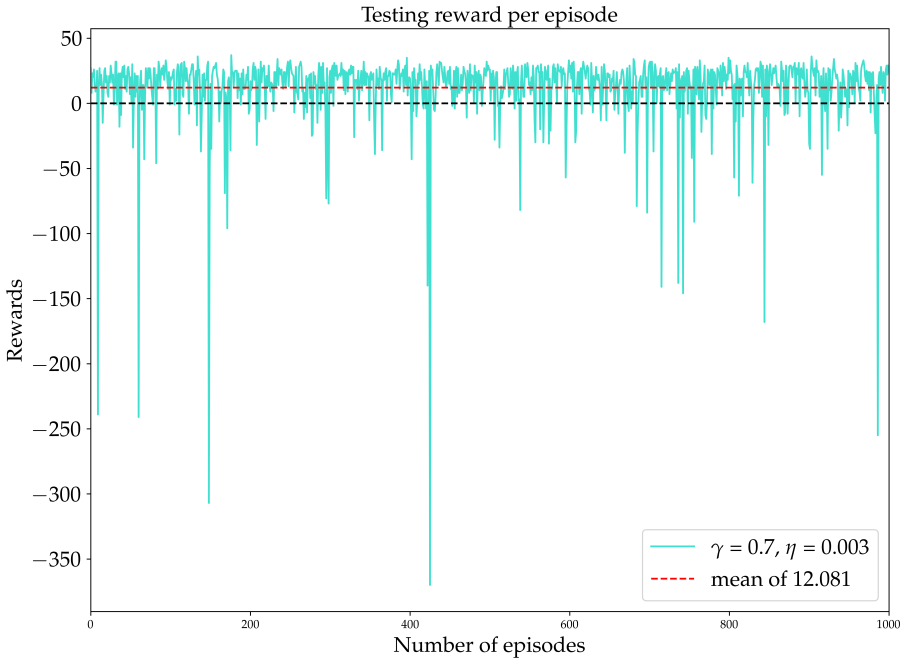


Figure 20: Testing reward for the best model for 1000 episodes

8.7 Identification Of Ride-Sharing Opportunities

state	South	North	East	West	Pickup1	Dropoff1	Pickup2	Dropoff2
12	-2.03535	-1.29254	-1.51108	-1.29455	-0.3654	-10.29257	-0.3654	-10.29232
19094	2.6525	1.12163	1.08349	2.65245	4.56563	-6.3475	4.56563	-6.34749
22125	-2.69679	-3.15744	-2.6968	-2.7096	-2.12098	-11.69652	-2.12098	-11.69652

Figure 21: Identification of similar action-values based on action "Pickup1" filtering

state	South	North	East	West	Pickup1	Dropoff1	Pickup2	Dropoff2
17087	11.7936	21.24	21.24	21.24	18.8	36	18.8	36

Figure 22: Identification of similar action-values based on action "Dropoff1" filtering

state	South	North	East	West	Pickup1	Dropoff1	Pickup2	Dropoff2
0	0	0	0	0	0	0	0	0
1	-0.89389	0.24288	0.24288	0.24288	-8.75712	-8.75712	1.5536	-8.75712
2	-1.64456	-0.80573	-0.8057	-0.8057	-9.8057	-9.8057	0.24288	-9.8057
3	-2.85252	-2.31565	-2.31565	-2.31565	-11.31565	-11.31565	-1.64456	-11.31565
4	0.24288	1.5489	1.5536	1.5536	-7.4464	-7.44656	3.192	-7.4464
5	-0.8057	0.24288	0.24288	0.24288	1.5536	-8.75712	-8.75712	-8.75712
6	-1.18856	-0.23921	-0.23605	-0.2357	0.95536	-9.23569	0.95538	-9.23569
7	-1.24629	-0.31964	-0.42667	-0.30776	0.86531	-9.30467	-0.30775	-9.30772
8	-1.32921	-0.41154	-0.41154	-0.41182	0.73561	-9.4074	-1.37324	-9.41151
9	-0.14895	1.06797	1.06797	1.06793	0.98376	-7.93203	2.58497	-7.93204
10	-1.64456	-0.8057	-0.8057	-0.8057	0.24288	-9.8057	-9.8057	-9.8057
11	-1.24812	-0.59161	-0.30778	-0.31059	-1.29171	-9.30775	0.86531	-9.30775
12	-2.03535	-1.29254	-1.51108	-1.29455	-0.3654	-10.29257	-0.3654	-10.29232
13	-2.08698	-1.36085	-1.35873	-1.35873	-0.44841	-10.35859	-2.08698	-10.35871
14	-0.26748	0.95128	-3.1463	0.95538	-0.7362	-8.04463	2.44423	-8.05417
15	-2.85252	-2.31565	-2.31565	-2.31565	-1.64456	-11.31565	-11.31565	-11.31565
16	-1.33156	-0.41486	-0.41155	-0.41155	-1.55735	-9.41151	0.73561	-9.41151

Figure 23: Excerpt of Q-table

9 References

Asynchronous advantage actor critic (a3c)-reinforcement learning-laymens explanation, Feb 2019. http://www.henrypan.com/blog/reinforcement-learning/2019/02/27/a3c-rl-layments-explanation.html.

T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, *Optuna: A next-generation hyperparameter optimization framework*, in Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2019. https://optuna.org/.

A. O. Al-Abbasi, A. Ghosh, and V. Aggarwal, *Deeppool: Distributed model-free algorithm for ride-sharing using deep reinforcement learning*, IEEE Transactions on Intelligent Transportation Systems, 20 (2019), pp. 4714–4727.

J. Bergstra and Y. Bengio, *Random search for hyper-parameter optimization*, The Journal of Machine Learning Research, 13 (2012), pp. 281–305.

A. Billard and D. Grollman, *Imitation learning in robots*, Jan 1970. https://link.springer.com/referenceworkentry/10.1007/978-1-4419-1428-6_758#:~:text=Definition,skillsperformedbyanotheragent.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, CoRR, abs/1606.01540 (2016).

F. Cardenoso Fernandez and W. Caarls, *Parameters tuning and optimization for reinforcement learning algorithms using evolutionary computing*, in 2018 International Conference on Information Systems and Computer Science (INCISCOS), 2018, pp. 301–305.

I. Caspi, G. Leibovich, G. Novik, and S. Endrawis, *Reinforcement learning coach*, Dec. 2017. https://intellabs.github.io/coach/.

J.-H. Chen, *Actor critic algorithm*, May 2018. https://www.slideshare.net/zhihua98/actor-critic-algorithm?from_action=save.

Y. Chen, A. Huang, Z. Wang, I. Antonoglou, J. Schrittwieser, D. Silver, and N. de Freitas, *Bayesian optimization in alphago*, 2018.

A. Choudhary, *Deep q-learning: An introduction to deep reinforcement learning*, Apr 2020. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

M. Claesen, J. Simm, D. Popovic, Y. Moreau, and B. D. Moor, *Easy hyperparameter search using optunity*, 2014. https://optunity.readthedocs.io/en/latest/#.

T. G. Dietterich, *Hierarchical reinforcement learning with the maxq value function decomposition*, Journal of artificial intelligence research, 13 (2000), pp. 227–303.

R. Dorfman and E. Ben-David, *Advanced topics in reinforcement learning (048716)*, 2018. https://github.com/eyalbd2/Deep_RL_Course.

M. Gupta, S. Aravindan, A. Kalisz, V. Chandrasekhar, and L. Jie, *Learning to prune deep neural networks via reinforcement learning*, 2020.

L. Hertel, P. Baldi, and D. L. Gillen, *Quantity vs. quality: On hyperparameter optimization for deep reinforcement learning*, arXiv preprint arXiv:2007.14604, (2020).

A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Stable baselines*. https://github.com/hill-a/stable-baselines, 2018.

A. Irpan, *Deep reinforcement learning doesn't work yet*. https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018.

A. Juliani, *Making sense of the bias-variance trade-off in (deep) reinforcement learning*, Feb 2018. https://link.medium.com/0a5tI3dambb.

R. Liessner, J. Schmitt, A. Dietermann, and B. Bäker, *Hyperparameter optimization for deep reinforcement learning in vehicle energy management.*, in ICAART (2), 02 2019, pp. 134–144.

H. Mao, M. Alizadeh, I. Menache, and S. Kandula, *Resource management with deep reinforcement learning*, in Proceedings of the 15th ACM Workshop on Hot Topics in Networks, 2016, pp. 50–56.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous methods for deep reinforcement learning*, 2016.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, *Human-level control through deep reinforcement learning*, Nature, 518 (2015), pp. 529–533.

S. Paul, V. Kurin, and S. Whiteson, *Fast efficient hyperparameter tuning for policy gradients*, CoRR, abs/1902.06583 (2019).

Z. SALLOUM, *Policy based reinforcement learning, the easy way*, Jan 2020. https://towardsdatascience.com/policy-based-reinforcement-learning-the-easy-way-8de9a3356083.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver, *Prioritized experience replay*, 2016.

M. Sewak, *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*, Springer Publishing Company, Incorporated, 1st ed., 2019.

R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2nd ed., 2018.

J. TORRES, *Deep q-network (dqn)*, Oct 2020. https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c.

H. van Hasselt, A. Guez, and D. Silver, *Deep reinforcement learning with double q-learning*, CoRR, abs/1509.06461 (2015).

N. Van Huynh, D. N. Nguyen, D. T. Hoang, and E. Dutkiewicz, *Jam me if you can: Defeating jammer with deep dueling neural network architecture and ambient backscattering augmented communications*, IEEE Journal on Selected Areas in Communications, 37 (2019), pp. 2603–2620.

T. Verma, P. Varakantham, S. Kraus, and H. C. Lau, *Augmenting decisions of taxi drivers through reinforcement learning for improving revenues*, AAAI Press, 2017.

Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, *Dueling network architectures for deep reinforcement learning*, in Proceedings of The 33rd International Conference on Machine Learning, M. F. Balcan and K. Q. Weinberger, eds., vol. 48 of Proceedings of Machine Learning Research, New York, New York, USA, 20–22 Jun 2016, PMLR, pp. 1995–2003.

Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye, *Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach*, in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018, pp. 905–913.

N. Yaraghi and S. Ravi, *The current and future state of the sharing economy*, Available at SSRN 3041207, (2017).