ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Autonomous and Adaptive System

DESIGN AND IMPLEMENTATION OF A REINFORCEMENT LEARNING FRAMEWORK FOR IOS DEVICES

CANDIDATE
Dott. Alessandro Pavesi

SUPERVISOR
Prof. Mirco Musolesi

Academic year 2020-2021
Session 3rd

This book is dedicated to the future. Whatever future it may be,
don’t lose the fire.

Abstract

Reinforcement Learning is an increasingly popular area of Artificial Intelligence. The applications of this learning paradigm are many, but its application in mobile computing is in its infancy. This study aims to provide an overview of current Reinforcement Learning applications on mobile devices, as well as to introduce a new framework for iOS devices: Swift-RL Lib. This new Swift package allows developers to easily support and integrate two of the most common RL algorithms, Q-Learning and Deep Q-Network, in a fully customizable environment. All processes are performed on the device, without any need for remote computation. The framework was tested in different settings and evaluated through several use cases. Through an in-depth performance analysis, we show that the platform provides effective and efficient support for Reinforcement Learning for mobile applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reinforcement Learning is becoming the new paradigm to approach some real-world problems. It is new to the general public, that might have heard about Machine Learning instead. Reinforcement Learning is the way an agent can learn from the experience, like us. Usually, this paradigm is applied to simulators that implement conditions similar to those of the final real-world into which the agent will be deployed. In this study, we have developed a Reinforcement Learning Framework to build a custom environment with which to train different agents. Everything working on Apple smartphones, tablets, and computers. The project consists of a Swift Package for the Apple Ecosystem which includes a fully customizable environment, two tested Agents, and the possibility to train the algorithm on an actual generation of iPhones and iPads, including interoperability with the Mac ecosystem.

In the literature, there are a lot of different projects that try different RL algorithms in different environments, or tasks simulators. For example, the OpenAI Gym [3] provide a lot of simulators, games, and software to train an agent in those environments; and it increases a lot the study and utilization of RL algorithms in the literature. As for Machine Learning, Reinforcement Learning needs a lot of power, both to train the agent and to run an environment that can be a simple or complex one. On the other hand, some projects try to limit energy consumption to run Machine Learning algorithms into small, low-powered devices, this is the sector of TinyML [21]. This branch works with very very limited boards, more similar to embedding systems than a smartphone that has a lot of power, especially the newest models, but needs to lower energy consumption as much as possible. The devices can produce a lot of data and this induces companies to integrate Machine Learning into these systems to allow the developer to use the device data, for example providing packages to deploy ML models directly on-phone ([13], [14]). Regarding Reinforcement Learning on a mobile device, there is a project called AndroidEnv [22] that exposes an Android simulator to be used as a Reinforcement Learning environment. Similar projects are not so mainstream the same idea has not been developed for Apple yet. So to my knowledge, this is the first research on the feasibility of training a Reinforcement Learning algorithm directly on Apple devices, and the first study about creating a framework for RL in the Apple Ecosystem. The package is developed in Swift, the programming language devised by Apple and used to replace the old Objective-C language. The development of a framework of Reinforcement Learning in the mobile sector had no real importance in the past basically because of two factors: the first is that Reinforcement Learning has gained popularity only in the last decade, and the second is the difficulty of training a Neural Network on so small devices as the mobile ones. Thanks to new technology and new hardware these two problems have been overcome and it's now time to bring Reinforcement Learning to the next step.

Building a framework for Artificial Intelligence with the intent of running it into smartphones could be a paradox because of the large memory footprint the neural network uses, the energy consumed to train a network, and the general use of a large amount of data for training. In the case of Reinforcement Learning, the case becomes more difficult because the training is not done with predefined data but the Agent needs to interact with the environment and learn through that interaction. This different formalization of the problem flow removes an important problem of Artificial Intelligence, as mentioned before, that of using a lot of predefined data to train the network, but how? The approach is through interaction with the environment, but talking about environment and interaction in the same sentence where we are talking about smartphones seems a little odd. We can think of the environment of a smartphone as of the smartphone itself, with its characteristics and properties, like the amount of battery remaining, or the brightness of the screen. The actions needed to interact with the environment and learn how it behaves in response are limited by the API distributed by Apple and the restriction of the Apple Developer Program.

Reinforcement Learning agents learn through interaction with the environment, but if the environment is built to evince the user behavior then the user could be the environment to test. This idea imposes to talk about the implication of the use of this package in terms of Ethics and Privacy. Ethics because a package that allows learning about its user's behavior, partially or completely, must be used ethically, without developing applications that can be used to study or to spy on someone. Privacy is highly correlated to ethics, it is one of the first things to be attacked when we talk about our devices. There are European rules [24] that try to limit the non-ethical use of the new world of Big Data, in which this project and Machine Learning can be included. We will talk about it.

I want to point out that this study is about iOS devices, but the package is built and developed to be used on all the devices that support Swift. This means that my package could be the first implementation of a Reinforcement Learning framework for Apple devices that can be used to develop applications for the devices and not only for research purposes.

Chapter 2

Background

Reinforcement Learning is a branch of Artificial Intelligence that uses the interaction between an Agent and an Environment to train the agent and let it learn how to behave, and at the same time improve a cumulative episode prize called *cumulative reward*. Thanks to this formalization of the problem it is becoming the third basic Machine Learning paradigm besides Supervised Learning and Unsupervised Learning.

Before deep-diving into how Reinforcement Learning works, let's have a short introduction to how we arrived at formulating Reinforcement Learning theory. The two most common paradigms are supervised and unsupervised, which are based respectively on a dataset of labeled data and unlabelled data. Both need a dataset, which is usually composed of a huge amount of data, collected and pre-processed before the algorithms see them. This is the main difference between these two archetypes of Machine Learning and the type used in this research. The process of retrieving data for supervised and Unsupervised Learning isn't correlated to the algorithm that will learn with it, the algorithm takes data and trains itself to improve its labeling skill for the Supervised Learning and the prediction skill for the unsupervised one. In the case of Reinforcement Learning, the data used to train the algorithm are created by the agent that interacts with the environment, but that is possible only because the training algorithm is based on rewards and not on data. We will clarify this phrase later: let's start understanding the basic thing.

2.1 Machine Learning

Machine Learning is the study of algorithms that learn and improve themselves automatically with the use of data, usually called experience. It includes all the algorithms in the field of Artificial Intelligence that have a model and use training data to predict or act without explicit programming. Its applications are manyfold: nearly every field in which data are created during normal execution can be a research field for Machine Learning. The data needed to train an ML algorithm can belong to any source, from marketing data such as sales or investments to images captured by cameras of autonomous cars; data are the most important part of Machine Learning algorithms. As we have already said, data are created and cleaned before being introduced into the algorithms, whether they are supervised or unsupervised. This implies that data can be created by processes not directly in the field of Artificial Intelligence, but that they can be every data that after a good pre-processing can be used by algorithms. Talking about Machine Learning algorithms, they are general, they can be applied to every data, they build a model to predict or act, and they are not programmed to do so, they are based on data which are their only source of learning. The learner's objective is to generalize from the experience, called *training data*, and perform accurately on new unseen data, called *test data*.

Trying to formalize this approach we can say that a program learns some type of tasks T from experience E, w.r.t. a performance measure P, if the performance increase with experience E.

Tasks

The tasks can be of any kind. The most common Machine Learning task is *Classification* in which the program, given an input, is asked to assign it to a category among many. Another is *Regression*, in which the program tries to predict a numerical value given some input, or *Structured output* tasks in which the output is a vector with strong internal correlation. Another important example is *Density estimation*, where the program is asked to learn a function approximating a probability density function or probability mass function.

Performance Measure

The Performance Measure is fundamental to evaluate how the agent improves its abilities for each task available. The measure is specific to each task, and different tasks ought to be evaluated with different measures. For classification tasks, the most common measure is *accuracy*, which is the value of correct output w.r.t. the total number of examples. On the contrary, the *error rate* is the value of incorrect output, w.r.t. the total number of examples. For Density estimation tasks we must use a metric that returns a continuous-valued score for every input. Usually the average log-probability values w.r.t. some inputs. The program is trained with a training dataset with some defined properties, but we want to use it with unseen data to allow it to generalize. Even if the performance measurement is done with a test dataset, fresh new data the learner has never seen must be processed, as it happens to the agent in the real world.

Experience

The Experience is the collection of the data that are used to learn. In the introduction we talked about the creation of data, the different types of algorithms divided into unsupervised and supervised. *Unsupervised Learning algorithms* use a dataset containing a lot of features and learning properties that can be useful to better understand the structure of the input dataset. The goal is to learn something that we as humans can find useful. We can define the kind of information to retrieve, but the process and the knowledge are retained by the algorithms. In the case of *Supervised Learning algorithms*, the dataset can contain the same number of features but it incorporates a label (or a target) consisting of what the algorithm has to predict at the end of the learning phase. In this case, the learner has the information to compare with its prediction, and can therefore improve itself. But these two are only the most common type of Machine Learning algorithms. Another interesting example is the *Semi-Supervised Learning* in which some data include the label and some others do not. This is another algorithm where the experience is contained into a fixed dataset, the world of Reinforcement Learning breaks down this barrier and opens to a more natural way of learning.

2.2 Reinforcement Learning

The idea of learning by interacting with the surrounding environment is fascinating, it reminds us of the evolution of our ancestors or, closer to us, of how children learn. Learning by interacting is the most natural way of learning: trial and error are how we approach a new situation and acquire useful information for the next time we are in the same position. If we keep experimenting with the, we can try to learn the cause-effect relationship between our actions and the effects those actions have on the environment. We are experimenting with ways to influence what happens through our behavior. *Reinforcement Learning is a computational approach that learns from interacting*. The phase of interaction needs to be guided by a goal, something the agent needs to reach, or to do, that concludes the interaction phase with some sort of failure or success. The approach is goal-oriented.

2.2.1 Intro to Reinforcement Learning

Reinforcement Learning is a learning process on how to associate actions to a situation with the scope of maximizing a numerical reward signal. This field of Artificial Intelligence is different from the other two more famous techniques, Supervised Learning, and Unsupervised Learning, from a different point of view. The learning agent doesn't know the target (or targets) of the simulation, nor the action it will take at a certain moment: the agent has to discover which action yields the highest reward or brings to a situation in which the reward seems better. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. We am going to describe the two most important features of RL, the trial-and-error approach and delayed reward. I wrote before that the learning agents have to build some connections between a situation and the possible actions to take but, as it often happens, they can have a limited vision of the environment. To overcome this limit the agent is sometimes provided with the entire observation of the environment, but in practice, this idea works only with easy examples. So, hypothetically, to learn how to maximize the final reward the agent needs to explore all the situations and all the actions and then choose the best action for each situation that leads to the maximum reward. In a real-world example, this can take too much time or too much computational energy to be effectively implemented. Because the agent's capability of seeing the world is limited, it will need to learn to explore the environment and take some numerical values to mark the best way to reach its end. Until now we haven't dealt with how the actions are chosen by the agent. We know that the actions are chosen concerning the final reward but how? While working at selecting an action, the agent chooses the one with a most likely future reward, which of course is calculated internally. This procedure is called *Exploitation*, and it uses the estimation of the value to find the best action to follow. The prediction of future reward depends on previous experience with that situation when action was selected and a reward was received, so there is a valid prediction only if the agent has already selected an action. By using this definition, the act of selecting an action is like a closed ring: if we can have a prediction of future reward only for actions that have been previously chosen, which has been previously visited, there can't be any improvement. The solution is *Exploration*, which is a technique that allows the agent to explore the environment without following the max future reward action. Now we can define the Exploration and Exploitation problem of Reinforcement Learning as the necessity of finding the trade-off between the two. To reach the goal of obtaining a lot of rewards the agent must prefer the actions that he knows to be effective thanks to the past, but to discover such actions he needs to exploit the environment making random choices and to try actions that have not been selected before. The Explore part uses the knowledge of the agent while the Exploit part makes it discover a new way that can be better or worse. The failure in exploiting is probable. The agent has to try different actions to gain experience and build its knowledge of the environment to finally reach the goal. Every Reinforcement Learning agent has an explicit goal that it has to pursue within an environment, using the information that it has. The information can be the whole environment or only a part of it.

We had an introduction of Reinforcement Learning elements but those elements interact with each other through well-defined time steps. In a single timestep, an interaction includes the process of gathering information from the environment and giving it to the agent that selects and carries out the action in the environment. The timestep ends with the retrieving of the reward and the updating of the agent's knowledge. The timestep can be infinite, where the agent has to act in a continuous environment or can be grouped into episodes. The latter technique is the most used one, defining a maximum number of timesteps that can be reached before the episode counter reaches, resetting the environment, and restarting the episode. This type of time organization is useful in different ways, for example, when the agent can be used within a simulator, to gain time during the training, or to train different agents at the same time.

2.2.2 Fundamentals of Reinforcement Learning

It's possible to identify three fundamental elements in a Reinforcement Learning system: a **policy**, a **reward signal** and a **value function**. We can include in this list also a model of the environment required by a certain type of system.

Policy

The policy describes the learned behavior of the agent at a given time. It is a mapping from a given state to the actions to be taken in that state, for each state observed by the environment. The policy can be implemented by a simple function or lookup table or, in a more complex way, by neural networks or search systems. The output of this mapping is usually a list of probabilities for each action. The Policy alone determines the behavior of the agent.

Reward Signal

The Reward Signal is the definition of the goal of a Reinforcement Learning problem. At each timestep, the environment gives back a numerical reward to the agent, and the only purpose of the agent is to maximize the total reward it receives over the long run. The numerical reward is interpreted as an indicator of a good or bad event. The reward is used to change the policy. If the agent with a low reward is given a state and an action, the probability of that action must be lowered to learn from that step. This change can bring to a future situation in which the agent will choose a different action if presented again with the state in which it got a lower reward.

Value Function

The Value Function can be confused with the reward signal because its output is similar to the reward for a timestep, but it is different and equally important. The Value Function is the indicator of how good it is to be in a given state w.r.t. in the long run. Usually, it is implemented in the form of the total amount of reward the agents predict to accumulate in the future starting from a given state. Of course, this measure is directly correlated to the Reward, since without reward there could be no values, so why use that? Because of how the action is chosen, action choice is based on the value function, because it is the most accurate indicator of how good the agent is doing to reach the goal of maximizing the total reward. The Value Function is updated at each time step and can be updated for the entire lifetime of the system, this is a major point of Reinforcement Learning.

Continuous Training

We keep learning during our entire life, the environment around us keeps evolving, because our mind grows and changes, and learn from everything we can feel. These are the basics of life. Reinforcement Learning has the potential of learning continuously from interaction thanks to its inner formulation. It is the technique most similar to us in Artificial Intelligence, as far as we know.

2.2.3 Tabular Methods

Bandit Problem

To start with the simplest form of Reinforcement Learning it's possible to think of an environment in which the state is always the same. The K-armed Bandit problem is a standard example in which the agent is asked to choose among k different options, each option returns a reward chosen from its stationary probability distribution. The scope is to maximize the total reward over a predefined time range. The agents start to play and keep track of the reward given by each action, the mean or the sum of the reward is the *value* of that action. If we denote the timestep as *t*, the action chosen at that timestep A_t, the reward R_t, then the value is defined as (for an arbitrary action *a*):

$$q_*(a) = \mathbb{E}[R_t|A_t = a]$$

This is called *Optimal Value*. Hypothetically, if we knew the value of each action a-priori it would be easy to solve the Bandit Problem, but of course, the agent doesn't know anything at the start and has to build its knowledge. The *Estimated Value* of an action *a* at timestep *t* is denoted as $Q_t(a)$ and the goal is to approximate as best as the agent can the optimal value. To calculate the estimated value the agent needs to interact with the environment and to do so it uses its policy.

As already described, the agent needs to find a trade-off between Exploration and Exploitation. Exploitation follows the *greedy* policy in which the action with maximum value is chosen at each timestep. As an alternative to include the exploration part the $\epsilon - greedy$ policy is one of the most important. It is used to define a value that describes which percentage of non-optimal actions will be chosen. This is mandatory to update the non-optimal actions the agent will never choose and find new ways of retrieving higher rewards. The concurrent use of exploration and exploitation can influence the duration of an episode. For example, if the agent has a single life in a game and during exploration it chooses an action that causes its death, the episode will stop immediately without reaching the total allowed timestep. So it is possible to say that exploration can be pejorative in most cases but the agent needs it to better approximate the optimal values.

There are also disadvantages in using an aggressive $\epsilon - greedy$ policy: for example in an environment with a lot of noise the exploration can increase the number of timesteps necessary to reach good performance a lot.

The $\epsilon - greedy$ policy is only an example among different and more complex policies, but it is by far the most used.

Action-value Methods

We talked about how to calculate the sum or the mean of the rewards given and action. So a fraction with the sum of rewards when an action *a* is chosen before the current timestep *t* and the number of time that action was chosen.

$$Q(a) = \frac{\sum_{i=1}^{t-1} R_i * \mathbb{I}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{I}_{A_i=a}}$$

The simplest policy follows the greedy action that in this case is evicted by the highest estimated value obtained by the previous formula.

$$A_t = \arg\max_a Q_t(a)$$

Exploiting the knowledge of the agent is good to know if the current strategy is good, using an $\epsilon - greedy$ policy is easy to implement due to the small modification to this approach and allows to explore.

Incremental Implementation

The necessity to evaluate a mean at each timestep could bring to a situation that the memory used during the process and the required computational power would increment to unsustainable levels. A better approach to limit the consumption of memory is to keep an incremental implementation of the mean that doesn't require all the data. An interesting property of this implementation is that the value update process is made by small, constant computational requirements to process each new reward.

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

In this formula new parts are presented:

- the *target* is a predicted value that presumably goes towards the goal.
- the *error* defined as $[Target - OldEstimate]$ is the correction of the value calculated for the current timestep.
- the *step size* is a parameter to give weight to the importance of that timestep update w.r.t. the total amount. Used preferably in an incremental method.

The step size is important because in real-world Reinforcement Learning problems the reward probability changes over time and there is the necessity to assign different weights to different updates. In such case it is preferable to assign higher weights to recent rewards than to past rewards, and lower weights to long-past rewards.

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$

Obviously not all the values can be assigned to the step size, some conditions are required to assure convergence:

$$\sum_{n=1}^{\inf} \alpha_n(a) = \inf \qquad \sum_{n=1}^{\inf} \alpha_n(a) = \inf$$

The first assure that the steps are large enough to overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps will become small enough to assure convergence. This points to the not convergence about the usage of constant step size.

Gradient Bandit Algorithm

The methods to estimate the action values aren't the only ones, it is possible to learn a preference for each action evaluated by percentage values. The *preference* for an action *a* $H_t(a)$ involves the *soft-max* function:

$$Pr\{A_t = a\} = \pi_t(a) = \frac{\epsilon^{H_t(a)}}{\sum_{b=1}^{k} \epsilon^{H_t(a)}}$$

Then, the probabilities updates become:

$$H_{t+1}(A_t) = H_t(a) + \alpha(R_t - \hat{R}_t)\pi_t(a)$$

where $\alpha > 0$ is a step-size parameter, and $\hat{R}_t$ is the average of all the rewards up through and including time t, which can be computed incrementally.

Contextual Bandit

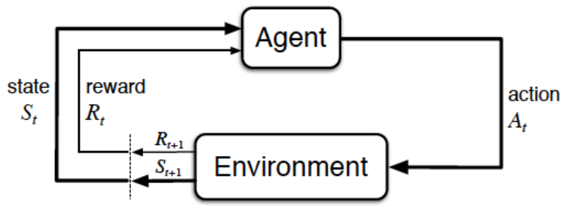
Until now we have talked about a very specific version of the Bandit Problem, in which the situation, or *state*, stays the same, and the only thing that changes are the reward probabilities. However, in general, in Reinforcement Learning tasks, there is more than one state and the goal of the agent becomes to learn an associative mapping from situation to the actions that best fit those situations. An example is still the bandit problem but with the modification when the agent selects an action that changes the state; it can also be seen as a nonstationary bandit problem. To build a correct *Contextual bandit problem* it is necessary to identify the situations so the agent can distinguish them and study a different policy for each state. These tasks are also called the Associative search task because it involves trial-and-error learning to search for the best actions and the association of these actions with the state in which they are best. The change in state after the chosen action is executed in this case is random, there is no correlation between the current state and the last one, or the future one; this is the main limit to this kind of task, the *Full Reinforcement Learning* involves an effect of the current actions to the next state.

2.2.4 Finite Markov Decision Process

A formalization of the tasks seen until now is the finite *Markov decision process* or finite MDPs, which involves evaluative feedback and a correlation between states. MDPs are a classical formalization of sequential decision making where actions influence subsequent situations through immediate and future rewards, there's the need to trade-off these two kinds of rewards. Given this difference, the value for action is derived not only by the action evaluated but also by the state of the agent.

Agent-Environment Interface

MDPs are intended to be a simple framing of the problem of learning from interaction to achieve a goal. The agent is the learner and decision-maker. The thing with which it interacts, which includes everything outside the agent, is referred to as the *environment*. The agent chooses actions, and the environment responds to these actions by presenting new situations to the agent. The environment also provides rewards, which are special numerical values that the agent seeks to maximize over time through its choice of actions.



In more detail, the agent and environment interact at each of a series of discrete-time steps *t* = 1, 2, ... At each time step, the agent receives some representation of the state S_t of the environment and chooses an action A_t based on that representation. The agent receives a numerical reward R_t and finds itself in a new state, S_{t+1}, a one-time step later, in part as a result of its action. As a result of the MDP and agent working together, the following sequence or trajectory is created:

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, ...$$

The sets of states, actions, and rewards (S, A, and R) in a finite MDP all have a finite number of elements. In this case, the random variables Rt and St have discrete probability distributions that are only dependent on the previous state and action. That is, given specific values of the preceding state and action, there is a probability of those values occurring at time t for specific values of these random variables:

$$p(s', r|s, a) = Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

for all states, rewards and actions allowed. The MDP's dynamics are defined by the function p. The dot over the equals sign in the equation reminds us that it is a definition (of the function p in this case) rather than a fact that follows from previous definitions. The probabilities given by p completely characterize the *dynamics* of the environment in a Markov decision process. That is, the probability of each possible value for St and Rt is determined solely by the immediately preceding state and action, $S_{t-1} and A_{t-1}$, and not by previous states and actions. This is best viewed as a state-imposed constraint rather than a constraint on the decision-making process. The state must include information about all aspects of the previous agent–environment interaction that differs for the agent. If it does, then the state is said to have the *Markov property*.

It is possible to compute the expected rewards for state–action-next state triples as a three-argument function:

$$r(s,a,s') = \mathbb{E}[R_t|S_{t-1}=s, A_{t-1}=a, S_t=s'] = \sum r\frac{p(s',r|s,a)}{p(s'|s,a)}$$

The MDP framework is abstract and flexible, and it can be applied to a wide range of problems in a variety of ways. Time steps, for example, do not have to be fixed intervals of real-time; they can be arbitrary successive stages of decision making and acting. Low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to eat lunch or go to graduate school, are examples of actions. Similarly, states can take many different forms. They can be completely determined by low-level sensations, such as direct sensor readings, or by higher-level and abstract descriptions of objects in a room.

The MDP framework is a significant simplification of the problem of goal-directed learning through interaction. It proposes that, regardless of the details of the sensory, memory, and control apparatus, and whatever goal one is attempting to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal representing the agent's choices (the actions), one signal representing the basis on which the choices are made (the states), and one signal defining the agent's goal (the rewards). This paradigm may not be enough for representing all decision-learning issues, but it is extensively helpful and relevant.

Goals and Rewards

In Reinforcement Learning, the agent's purpose or objective is codified in terms of a particular signal called the reward that travels from the environment to the agent. A prize is a simple number at each time step. The agent's informal objective is to maximize the overall amount of payment it receives. This involves optimizing long-term cumulative reward rather than immediate reward. One of the most distinguishing elements of Reinforcement Learning is the use of a reward signal to define the concept of a goal. Although expressing objectives in terms of reward signals may appear to be limited at first, it has shown to be flexible and broadly applicable in reality.

When teaching a robot how to escape from a maze, the reward is frequently 1 for each time step that passes before escape; this motivates the agent to escape as soon as feasible. To teach a robot to search and collect empty soda cans for recycling, one may offer it a zero reward most of the time and a +1 reward for each can be collected. It could also be a good idea to offer the robot negative reinforcement when it collides with anything or when someone screams at it. The natural rewards for an agent learning to play checkers or chess are +1 for winning, 1 for loss, and 0 for drawing and all non-terminal positions.

Returns and Episodes

We said that the agent's purpose is to maximize the cumulative reward it gets over time. How might this be formalized? If we represent the series of prizes received after time step t as $R_{t+1}, R_{t+2}, R_{t+3}, ...$, then what specific element of this sequence do we want to maximize? In general, we want to maximize the expected return, where the return, abbreviated G_t, is specified as a particular function of the reward sequence. In the most basic scenario, the return equals the sum of the rewards, this method is appropriate in cases where the last time step is a natural concept:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...$$

when the agent–environment interaction naturally breaks down into distinct subsequences, which we call *episodes*, such as gameplays. Moreover, each episode concludes in a special state known as the terminal state, which is then reset to a standard starting state or a sample from a standard distribution of starting states. Even if you consider episodes finishing in many ways, such as winning or losing a game, the next episode begins no matter how the previous one ended. As a result, all episodes may be thought of as ending in the same terminal state, with different rewards for different outcomes. This type of activity is referred to as an *episodic task*.

The return can be *discounted* to set more weights to the near reward and less to the long-term reward.

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

where γ is a parameter, $0 < \gamma << 1$, called the *discount rate* that determines the present value of future rewards.

This formula can also be used with the incremental version that becomes:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Policies and Value Functions

Estimating how beneficial it is for the agent to be in a given state usually involves *value functions*, and until now we have used the future reward estimated as the indicator. Though value functions are defined concerning *policies*.

A policy is formally defined as a mapping from states to probability of taking each potential action. $\pi(a|s)$ is the chance that $A_t = a$ if $S_t = s$ if the agent is following policy at time *t*.

The *value function* of a state *s* using a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For Markov Decision Processes is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t], \forall s \in S$$

This is called *state-value function fr policy* π

Taking an action involves evaluating the expected return from a starting state s_t taking an action a_t following a policy π. Called *action-value function for policy* π:

$$q_\pi(s, a) = \mathbb{E}_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k Rt + k + 1 | S_t = s, A_t = a], \forall s \in S, a \in A$$

The two value-function above are estimated using the experience the agent collect following the policy π

We name these types of estimating methods Monte Carlo methods because they include averaging across a large number of random samples of real results. For instance, if there are a lot of states, keeping separate averages for each one may not be feasible. Instead, the agent would need to keep v_π and q_π as parameterized functions (with fewer arguments than states) and tweak the parameters to reflect the observed results.

A key characteristic of value functions is that the following consistency condition exists between the value of s and the value of its potential successor states for any policy pi and any state *s*:

$$v_\pi(s) = \textstyle\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \forall s \in S$$

Optimal Policies and Optimal Value Functions

For finite MDPs we can define what is called an *optimal policy*, denoted by π_*. Thought, at the same time it's possible to define also the optimal *action-value function*, denoted q_*:

$$v_*(s) = \max_\pi v_\pi(s) \qquad\qquad q_*(s,a) = \max_\pi q_\pi(s,a)$$

$\forall s \in S.$

Once one has v_π, it is straightforwards to determine an optimal policy.

On-Policy and Off-Policy

The policy is what defines what action is to be chosen at a certain time, in a certain situation. There are different policies, we have seen some in the previous sections like the epsilon-greedy. It's possible to define two methods, *On-policy* methods attempt to evaluate or improve the policy that is used to make decisions. In contrast, *Off-policy* methods evaluate or improve a policy different from that used to generate the data.

2.2.5 Monte Carlo Methods

Monte Carlo methods are ways of solving the Reinforcement Learning problem based on averaging sample returns. We assume experience is divided into episodes, and that all episodes eventually terminate, no matter what actions have been selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense. The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns

Monte Carlo methods sample and average returns for each state-action pair much like the bandit methods The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the reward given after having taken an action in a defined state depends on the actions taken in later states in the same episode.

Predictions

Using Monte Carlo methods to predict and learn the state-value function, following a policy, means estimating it from experience and then averaging the rewards observed: with more experience it is possible to converge to the optimal value function. There are multiple choices on which data to use to average, for example, it is possible to take into account only the return of the first visit of the state (given that is used an episodic task, the first visit is intended for the single task), called *first-visit MC method* or taking into account every visit, *every-visit MC method*. First-visit MC has received the greatest attention, dating back to the 1940s. Every-visit MC naturally extends to function approximation. Both first-visit MC and every-visit MC converge to $v_*(s)$ as the number of visits to s goes to infinity. The estimations for each state are independent, which is an important aspect of Monte Carlo techniques. As in DP, the estimate for one state does not build on the estimate of any other state. In other words, Monte Carlo approaches do not bootstrap in the sense that we described it in the preceding chapter. It is worth noting, in particular, that the computational cost of determining the value of a single state is independent of the number of states. This can make Monte Carlo approaches particularly appealing when only one or a subset of states is required. Many sample episodes can be generated by beginning from the states of interest and averaging results from only these states while disregarding all others.

Estimation

To estimate the *state-action values* becomes important if there isn't a model of the environment, suggesting a policy. So, the goal of Monte Carlo Methods is usually to estimate q_*.

Here is the integration of the *policy evaluation* problem that have to estimate $q_\pi(s, a)$. The state-action pair is visited when the agent takes the action

a at the state s. The policy can become a problem if it is deterministic because it will always select the same action, meaning that the other state-action pairs will not be visited and so estimation is impossible. Though the problem is approached using Exploration.

Control

The *Monte Carlo Control* problem is to approximate optimal policy. At the simple stage this problem is divided in *policy evaluation* and *policy improvement*, the former is made as described so far, using experience and averaging the returns; the latter is done by making the greedy policy w.r.t. the current value function. The correction is made by:

$$\pi(s) = \arg\max_a q(s, a)$$

This is an approach that uses only the policy to change the environment and at the same time is also used to control, this is called *on-policy* method. In the *off-policy* methods these two functions are separated, the policy is used to generate experience but in this case, not all the experience is made by following it. An example is to use $\epsilon - greedy$ to behave or as a target policy.

2.2.6 Temporal-Difference Learning

Temporal-difference (TD) methods can learn directly from raw experience without a model of the environment's dynamics. These methods update estimates based in part on other learned estimates, without waiting for a final outcome (they *bootstrap*).

Predictions

Both TD and Monte Carlo methods use the experience to solve the prediction problem but Monte Carlo methods wait until the return following the visit is known, then use that return as a target for V (St). TD methods need to wait only until the next time step. At time t + 1 they immediately form a target and make a useful update using the observed reward Rt+1 and the estimate V (St+1). The simplest TD method makes the update:

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

The difference from Monte Carlo methods is the target, in MC the target is G_t, while in TD methods is $V(S_t) = R_{t+1} + \gamma V(S_{t+1})$. What is on the bracket is a sort of error and is called *TD error*.

Using this formula to update the value function is also called TD(0) or *one-step TD*.

The benefit of TD techniques over Monte Carlo methods is that they are done naturally in an online, completely incremental approach. With Monte Carlo techniques, one must wait until the conclusion of an episode, since the return is only known at that point, but with TD approaches, only one-time steps are required. Surprisingly, this is frequently a key aspect. Because some programs have extremely long episodes, deferring all learning until the end of the episode is too sluggish. Other applications keep working with no incidents. Finally, as discussed in the previous chapter, some Monte Carlo approaches must disregard or devalue episodes during which experimental actions occur, which can significantly deter learning. Because they learn from each transition regardless of what following actions are made, TD approaches are substantially less prone to these issues. TD(0) has been shown to converge to v_π for any fixed policy π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter declines according to the normal stochastic approximation requirements.

Sarsa

An example of On-Policy TD Control Methods is *Sarsa*, called before the use of the State, the Action, the Reward, the next State, and the next Action to produce an update.

As said, the method is on-policy so the update will follow the policy defined, to estimate the action value:

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

We use this formula to update after each step, also if the state is terminal (in that case the third part of the TD error is zero). This rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

Q-Learning

For the off-policy TD Control algorithm is *Q-Learning* which changes a bit the update rules:

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

In this formula, the TD error is made by changing the second element, here the action chosen isn't directly picked up by the policy but is the best action to be taken in the next state. Instead, the On-policy method uses the policy also to define the action to be done. An advantage of Q-Learning is to approximate directly q_*, independently from the policy followed, that maintains the utility of determining which state-action pairs are visited and updated.

The Q-Learning is at the base of Deep Q-Network, a special version of Q-Learning made with neural networks. This version can be used in continuous contexts thanks to the neural network used as an approximator instead of a matrix.

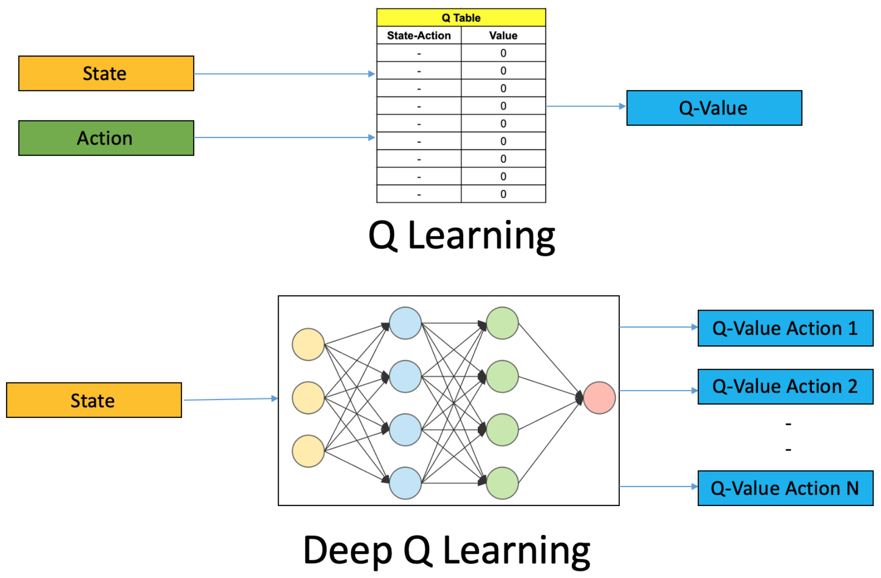


Figure 2.1: Q-Learning vs DQN [12].

The DQN has different variants, the Double DQN uses two DQN to correctly estimate the Q-Value, alternating their training during the learning phase. A similar version is the DQN with the Target Network, which uses the same network but is not trained to evaluate the future state during the updating. The Target network is updated not with normal training but after a defined number of steps the official network is copied into the target network.

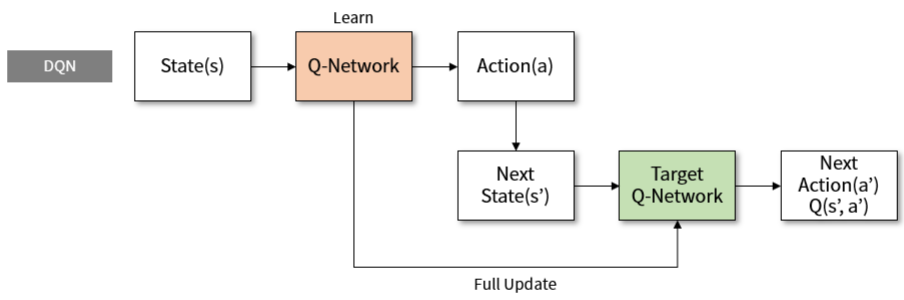


Figure 2.2: Deep Q-Network with Target Network [6].

2.3 Apple Ecosystem

The $AppleEcosystem^{tm}$ is not a product of Apple. The Apple Ecosystem cannot be purchased at the Apple Store, rather it is a user and lifestyle experience that comes as a result of owning many Apple devices. Consider this: when you obtain a rare set in your favorite role-playing game, you gain character perks. That is really what ecology is. Special features are available on your Apple devices that elevate the user experience due to interaction between Apple devices. Because no other firm covers the complete stack, the Apple ecosystem is better than that of competing brands like Android or Microsoft.

Microsoft may compete with Apple for the desktop, Google for mobiles and maybe a tablet, and Sony for games, but Apple has the vertical integration of delivering desktop, mobile, watch, table, and TV interfaces. Apple's ecosystem offering now outperforms the competition because they not only control the software that drives the hardware, but they also control the hardware itself. Apple now has the same hardware platform on all of its products, from small little AirPods that fit into your ears to desktops like your iMac, thanks to the launch of the M1 CPUs. Unlike other brands, Apple can customize the experience since every device has the same hardware architecture (licensed ARM instruction set but bespoke processor from Apple).

So, the Apple Ecosystem is a unique experience that you will get when you have a plethora of Apple devices from desktops and laptops to watches and wireless earbuds. Their unique differential factor is that they control the entire stack so they can provide you with a tailor-made experience. While other brands provide a subset of their services and cross-platform integrations, such as Zoom for meetings, Telegram for messaging, and Gmail for email, no one develops the full stack like Apple.

2.3.1 Swift

Swift is an extremely powerful and user-friendly programming language for iOS, iPadOS, macOS, tvOS, and watchOS. Swift is a general-purpose programming language that was designed with a contemporary approach to safety, performance, and software design principles in mind.

The Swift project's purpose is to provide the best accessible language for a variety of applications spanning from system programming to mobile and desktop apps, all the way up to cloud services. Most significantly, Swift is intended to make it easier for developers to write and maintain proper applications. To accomplish this, Apple believes that the most apparent approach to create a Swift code should also be:

- Safe: The most apparent approach to creating code should also be the safest. Undefined behavior is the enemy of safety, and developer errors should be identified before the software is released to the public. Swift may appear severe at times when choosing safety, but we believe that clarity saves time in the long run.

- Fast: Swift is designed to be a successor for C-based languages like C, C++, and Objective-C. As a result, Swift's performance for most jobs must be equivalent to those languages. Not only must performance be predictable and constant, but it must also be quick in brief bursts that require cleanup afterward. There are many languages with innovative traits; yet, being rapid is uncommon.

- Expressive: Swift leverages decades of advances in computer science to provide syntax that is easy to use while providing the contemporary capabilities that developers need. Swift, on the other hand, is never satisfied. We will continue to follow language improvements and embrace what works, developing to make Swift even better.

2.3.2 CoreML

Apple's CoreML is a new Machine Learning framework. Machine Learning may now be implemented in apps with just a few lines of code. Core ML allows you to incorporate a wide range of Machine Learning model types into your project. It supports traditional models such as tree ensembles, SVMs, and generalized linear models, in addition to advanced Deep Learning with over 30 layer types. Core ML makes use of the CPU and GPU to give optimum performance and efficiency because it is built on low-level technologies such as Metal and Accelerate. You can run Machine Learning models on the device, which eliminates the need for data to leave the device to be examined.

Core ML is compatible with a wide range of Machine Learning models, including neural networks, tree ensembles, support vector machines, and generalized linear models. The Core ML model format is required by Core ML (models with a .mlmodel file extension). Apple also offers several popular open-source models that are already in Core ML model format. You may download these models and use them in your app right now. Using Create ML and your data, you can train bespoke models to do tasks such as picture recognition, text extraction, and detecting correlations between numerical numbers. Create ML models are in the Core ML model format and are ready to utilize in your app.

On-Device Updates

You may customize an updatable model on the user's device at runtime using the Core ML framework. You may build a tailored experience for the user while keeping their data secret with this strategy. Instead of training, Apple refers to it as on-device customization. The purpose of these new APIs is to allow users to fine-tune an existing model based on their data. Apple's model training software for macOS is Create ML. It's ideal for quickly constructing basic models like image/sound/text classifiers. It uses Transfer Learning to shorten training times, similar to Turi Create, Apple's other teaching product. Create ML and Core ML were created to function in tandem: After training, you may save your model directly in Core ML format. There is no need to use a converting tool beforehand. A CreateML.framework is also available, which allows you to train models from a Swift script or Playground. It's crucial to note, however, that on-device training does not use the Create ML framework at all. Core ML is always used for on-device model customization.

Xcode

Xcode is Apple's integrated development environment (IDE) for developing programs for Apple devices such as the iPad, iPhone, Apple Watch, and Mac. Xcode includes tools for managing the complete development cycle, from app creation through testing, optimization, and submission to the App Store. Xcode is the only editor allowed to build applications for Apple devices, it is completely integrated into the MacOS system and has interesting capabilities. The Simulator included in Xcode allow to try the application directly on a different device, simulating the operating system selected, and include a set of commands to set some useful parameter of the phone (e.g. rotate the device, modify the volume or the location). Instruments provide tracking for different types of activity, for example, the CPU or memory usage, the memory leak, battery consumption, almost any kind of activity the device can do. It is perfect to easily see and keep track, with easy-to-read graphs of the app status and problems.

2.3.3 CoreMLTools

The *CoreMLTools* is a package of tools useful to create custom neural networks from scratch or convert existing neural networks into a compatible version for the Apple Ecosystem. Apple provides a lot of different pre-trained neural networks for different types of tasks such as Object Detection, Regression, or Natural Language Processing. Those networks are the most commonly used to create a mobile application so isn't strange that they don't provide a network for Reinforcement Learning. Moreover, the provided network can be really big and training, or fine-tuning the entire network on-device is out-of-scope. The necessity of creating a customized architecture for Reinforcement Learning is satisfied by the Python version of CoreMLTools that provides simple APIs to build a network from scratch. Moreover, it also allows you to select the layers that will be updated during on-device training.

2.3.4 Apple Hardware

All the iPhones and iPads since 2011 include a *Neural Engine*, a unique processor that speeds up Machine Learning models, but little is known about how this engine works. The Apple Neural Engine (or ANE) is a form of NPU (Neural Processing Unit). It's similar to a GPU, except instead of speeding visuals, an NPU accelerates neural network functions like convolutions and matrix multiplications.

In the last year, Apple has developed and distributed a new ARM processor made by itself that contains an updated version of the Neural Engine, the series of processor Mx. The various configurations of the processors are used in the MacOS devices so it's easier to develop an application and distribute them in all the Apple ecosystem without much work. This new opportunity allows the use of my package with the support of the Neural Engine when supported.

Chapter 3

Related Work

Reinforcement Learning has become more and more important in the last decades, its applications range from simple environments to complex simulations of physic problems or social simulation via multi-agents systems. For the sake of this research, the related work has to deal with a mobile implementation of a Reinforcement Learning platform or research about the feasibility of applying Reinforcement Learning to a mobile device. Despite my efforts, we have found few studies on the first kind of application, and they are different from what we have done. Use of Reinforcement Learning is applied to the use of the device, to simulate human behavior or to test the components. Furthermore, correlated to this study various projects are pursuing the application of Machine Learning and Reinforcement Learning into the common smartphones, for example, the kit provided by the two most important brands of smartphones.

3.1 Software

Generally speaking the application of Artificial Intelligence to mobile devices concerns the implementation of Machine Learning tools and methods, for example, TensorFlow [16] provides Tensorflow Lite, an open-source Deep

Learning framework for on-device inference. Using it, it is simple to deploy Machine Learning models on mobile, as well as fine-tune the models on-device. Tensorflow Lite is compatible with the most important mobile operating system, Android and iOS. It allows also the possibility to apply Federated Learning: to train global models on decentralized data. TFLite includes some pre-trained models for the most common tasks, for example, models for Natural Language Processing or Image Classification, as well as Regression and so on. Another example of a Machine Learning package is the Google ML Kit [14], which can also be used as well in both the most common O.S. and includes almost the same feature of TFLite, with models for Pose Estimation, Hand Written Classification, and more. Regarding the Apple ecosystem, besides other than the two above-mentioned possibilities, there is the Apple Core ML package [13] to deploy Machine Learning models on-device. Similar to the previous package the capabilities of this package ranges from sound analysis, to sound recognition or natural language processing. The field of TinyML [21] explores the types of models you can run on small, low-powered devices like microcontrollers. It enables low-latency, low power, and low bandwidth model inference at edge devices. The packages presented before are examples where the TinyML concepts are applied, they must run on a mobile device while consuming as little as possible. The real application of TinyML are those which runs in microcontrollers and embedded systems where the devices run unplugged on batteries for weeks, months, and in some cases, even years, while running ML applications on edge.

Frameworks for Machine Learning on mobile are limited yet, and those that allow applying Reinforcement Learning are even less. OpenAI Gym [3] is one of the most commonly used frameworks to try Reinforcement Learning on computers. In practice Gym is the standard for Reinforcement Learning framework, usually, all the new platform or new simulator is built to be compatible with it. There is not a version for mobile devices yet, but the structure of this project helped me to build this new RL framework.

Swift is a programming language developed by Apple but that is not only used by Apple devices, instead, it has been created to make developing apps easier but also to rejuvenate Objective-C in a new way.

3.2 Existing Projects

Machine Learning in its various sector is used for a long time in mobile devices. From the first application of Machine Learning the studies are more and more complex, for example in [7] the authors studies Mobile health (mHealth). It is considered one of the most transformative drivers for health informatics delivery of ubiquitous medical applications. To tackle the challenges of mHealth applications, they present an on-device inference App and use a dataset of skin cancer images to demonstrate a proof of concept. In [11] there is an application of Machine Learning to a novel approach to protecting mobile devices from malware that might leak private information or exploit vulnerabilities. The approach, which can also keep devices from connecting to malicious access points, uses learning techniques to statically analyze apps, analyze the behavior of apps at runtime, and monitor the way devices associate with Wi-Fi access points. The researcher studies real application and creates apps for the mobile device that helps the users and that are based on Machine Learning technique, as in [9]. The use of learning techniques for personalization ad-hoc for each user is a possible development using my RL package while regarding Machine Learning it is possible to use it for the same goal, as in [25]. In this article, the researcher presents a flexible Machine Learning approach for learning user-specific touch input models to increase touch accuracy on mobile devices. The model is based on flexible, non-parametric Gaussian Process regression and is learned using recorded touch inputs. Deep Learning, a subsector of Machine Learning, as well is applied on mobile devices. in [8] the authors give a review about the current projects, the limits, and the problems to overcome to improve the use of Deep Learning on mobile devices.

There are various studies about the application of RL on mobile devices: in the following sections, there are a few examples.

In [20] the authors present an open-source Reinforcement Learning (RL) research platform devised for the Android environment through a universal touchscreen interface. AndroidEnv enables RL agents to engage with a wide range of apps and services typically used by humans. In this project, the agent can use a simulated touchscreen to interact with the device and the operating system, aiming to learn how to use the device as humans do. The paper explains the work of applying the RL algorithms to an Android system, with the interaction limited to the use of the touchscreen. This is the main difference between this project and mine, the interaction with my framework can be at any level, from a low-level adjustment of setting to the high-level use of the touchscreen.

In [1] it is described a technique for automated GUI testing of Android apps based on Reinforcement Learning. They employ a Q-Learning-based test generation method to systematically choose events and explore the GUI of an application under test without the need for a previous abstract model. In this paper, Reinforcement Learning is used to test the Android GUI of applications without the agents needing a model. This approach is similar to the previous one, they are both testing the human-level interface to optimize the code below them.

In [10] the authors propose a system to be integrated with the Energy-Aware Scheduler (EAS), it organizes CPU hardware information into Energy Model which is used to improve CPU scheduling performance. This paper introduces the Learning Energy-Aware Scheduler (Learning EAS) which improves power consumption and solves the lack of adaptability of standard systems. This project works with low-level parts of the device, specifically at the kernel level to improve the performance of the device. This is a different example of RL application, but another example of how this field can be used in a different context.

In [4] the authors have approached the interesting problem of continuous sensing in smartphones. Continuous passive sensing via smartphone integrated sensors can quickly deplete the battery, interfering with other device functions. They offer a new adaptive sensing framework that uses Reinforcement Learning to optimize sensing time to increase energy efficiency in continuous mobile sensing applications. They characterize our adaptive sensing problem as a Markov Decision Process and adjust the sensing time of targeted sensor(s) dynamically so that they only operate in desired circumstances (e.g. collect accelerometer data only when the phone is moving). This paper can be applied to the framework I've built, given the possibility of using the integrated smartphone sensors to create the environment and so the state used by the agent.

The authors in [19] have introduced DeepAPP, a Deep Reinforcement Learning framework that builds a model-free prediction neural network using past app consumption data. This is another example of Reinforcement Learning use in improving the performance of a mobile device.

3.3 Comment

The branch of TinyML includes all those frameworks that have to work in small, energy-aware, low power systems. Smartphones are slightly included in this sector thanks to their power that nowadays is nearly the same as the power notebooks or some mid-level desktop computer. But the characteristic of being energy aware is important at the same level for TinyML systems and smartphones because also the latter have to consume the least possible amount of battery during their lifetime. Device brands are struggling with smartphone autonomy as much as for computers and tablets, trying to squeeze the most from the battery to let the device battery live longer. All these devices have some learning algorithm that studies the user habits to adopt the charging to them. This project can add a new way to do this. Regarding the studies we have quoted in this section, they are all applications of RL to mobile devices, from using it to test the GUI usability and performance to improve the low-level management of device resources. They all have one thing in common: the application of Reinforcement Learning to solve a problem or to optimize a process. AndroidEnv is the different one, it provides a platform to test the RL algorithms in a complete Android environment. Therefore it provides a platform with which the developer community can test their App or new ideas. It is similar to the idea described in this paper, providing a service for the developer community.

Reinforcement Learning is becoming the third most important type of Machine Learning, the potentiality of this type of algorithm is being discovered in these years. There are a lot more studies about Reinforcement Learning applied on-device, we have summarized only those which correlate to my paper, in my opinion.

Chapter 4

Design

In this study, the approach is based on those described in the Related Work, the concepts used conceptually designed as similar as possible to those to maintain a similar methodology and a continuity with State Of The Art research. The Apple Ecosystem helps interoperability of apps and packages over different devices, this simplifies the development of a single package of a Reinforcement Learning Framework for all the devices on the Apple Ecosystem. The project has the main purpose of studying the feasibility of applying Reinforcement Learning algorithms directly to Apple devices. As a secondary purpose, the package needs to be reliable, easy-to-use, and customizable. The main *elements* of a Reinforcement Learning Project are the **Agent** and the **Environment**, these concepts are also the main actors of the framework developed.

The environments usually seen in university literature can be of any kind, from simulations of robots (or part of them) to more abstract simulators to study the behavior of a multi-agent system, but also videogames, as the famous Atari collection included in the OpenAI Gym library [3], or more complex 3D games such as Starcraft [23]. All these applications of Reinforcement Learning share a similar concept of Environment, a class that elaborates the behavior of a pre-developed program, modified if possible in a discrete-time step program, and that returns the state of the environment as well as the reward assigned. Everything is elaborated internally, the behavior is defined a priori if we approach the game of Breakout [17] in the Atari package, the rules of the game are fixed, as well as the interface. Furthermore, the reward system is defined and well-known because of the explicit rules of the game that are set when a life is lost or when the episode ends. Actions are another example of fixed properties: in an Atari Game, for example, actions are the possibility to move in the cardinal directions. An Environment is a box that includes all these blocks, that are predefined and with which the Agent interacts. We approach a problem defined by other people.

Therefore the creation of a custom environment involves the development of a program that includes something to observe and from which to create the state, the list of actions allowed, and the reward system based on the goal we want to reach. In certain situations it can be a new game or a new simulator, in other situations it can be the set of sensor systems attached to a robot (the whole robot or just a part, like a robot's arm, it's only a matter of complexity) or a platform. In case the environment is directly the real world the definition of the environment rules may derive from the real world limits of the system. Therefore the reward is defined over a real-world parameter taking into account the environment limits, like the range of movement an arm can do, or the movement limits due to the size of the environment. The reward defines the purpose of the agent so it is strictly connected to the general characteristics of the environment.

4.1 Environment

The main part of building a new Reinforcement Learning framework is the definition of a new environment that contains observable properties, defined actions, and a reward system. The environment is what allows researchers to test their theories. It is the main part of this kind of study. If a researcher wants to study the possibility to use Reinforcement Learning algorithms to guide the exploration of a new world, he must create an environment (in this case it can be a simulator) of the new world adapting the rules to a new form with which it is possible to try the RL algorithms. The basic rules to adapt a problem for Reinforcement Learning is the theory (chapter 2.2), from there is possible to extract the basic rules and create a framework for developers that offers the possibility to simplify those basic rules, almost forgiving it, and adapt the problem to the RL algorithms. In this study, we propose this help to the developer regarding the Apple devices.

The package offers the possibility to create an infinite number of different environments thanks to the opportunity of including new Sensors developed by external people as well as Sensors that elaborate internally the information and return values not correlated with a physical feature. Moreover, the Actions can be used to characterize the agent's moves and they are responsible for the interaction of the agent with the real device. The Reward system as well is customizable, allows to define every kind of reward based on the other data.

The *environment* includes all those rules that characterize it and distinguish it, an environment is created because it is interesting to study its properties. The concept applied to Apple devices is converted into what can be studied with the information that can be retrieved from the device. A single device can provide a lot of information about the device itself but also information about the users, for example, it can provide information about their usual behavior, their appointments, and so on (see chapter 7.1, for a discussion about the ethic and privacy implication of this project). If the device itself can become the environment, then it is possible to unlock a lot of data that can be used to learn and help users with their lives. This definition includes the idea of a read-only environment created on the device from which we gather data, but to implement Reinforcement Learning it must be an editable environment in which the agent(s) operate and modify its characteristics.

It is possible to have an environment with a large number of observable characteristics, especially if it is based on the real world. An example can be an autonomous car: if the designer uses a lot of sensors or cameras then the data contain a lot of information, but for a self-driving car it might not need to use all of that information. For my research, we do not have a specific goal for the environment so we am building the environment as a customizable object that returns all the information needed by the agent to act and learn, without really knowing what the action and what agent are doing, neither what the environment produces.

In games like the ones included in the Atari collection, it is not possible, within the same game, to choose a different configuration of rewards or actions, or what to see of the game (usually the screen). The framework of these games allows the connection of different agents but not the modification of the environment, that is pre-developed. In the Apple devices, with my framework, it is possible to select the information needed, moreover, the actions and the rewards are delegated to the developer.

Therefore one of the main characteristics is *customization*, the Swift Package is developed for the developer so they can build their custom environment using parts already defined or implementing new ways of retrieving information, just as it is up to them the definition of the actions and the reward system.

Conceptually the environment has to read objects and return the data with a predefined structure, moreover, it has the goal of managing the interaction between the Agent and the actual device to modify its properties. So the agent chooses what action to make but only the environment knows how that action is made, the set of activities involved to do an action that will physically modify the device properties.

4.1.1 Discrete Time

Every algorithm we have seen in the previous Background chapter (number 2) uses the concept of the state as the view of the environment at a certain time instant, the principle of taking a photo of the environment and assuming it remains the same until the next photo is taken is an approach that uses the notion of *Discrete Time*. The same approach is implemented in this framework, the state is read at determined time instant and defines the values of the observable properties at that time, but more generally it describes how the environment evolves; it defines the flow of time, or more technically it can be viewed as the variable *T = 0, 1, 2, ...* that goes on indefinitely.

4.1.2 Action

An action is an object that is executed by the environment and chosen by the agent. There can be one or more actions based on the developer's implementations. An action must define what is the real action in the environment, thus the environment commands. They can be everything from modifying a property of the device to sending a notification to the user, the developer can use it without limits.

The actions are built by the developer to implement a way to reach the agent's goal. From the package point of view, the action can be anything but must be written following a standard structure.

The action concept is wide, it can set the brightness of the screen but also notify the user that something has happened, moreover it can be implemented to control an external object, for example, if the device is connected with a robot hand, there is nothing to limit this situation.

4.1.3 State

Above we have talked about the *State* that is built with the value of the observation made by the environment. The observation concern the values of internal device characteristics or external elaborated signals; the decision is made by the developer to fit the problem he wants to approach. To make this observation process as customizable as possible each observable property must be chosen or developed by the developer using a predefined structure. The State is composed of an arbitrary number of Sensors. The concept of the sensor can be misunderstood because of its name, but it does not apply only to the sensors of the device that read values like the battery or the brightness of the screen, but it can also be applied to everything that can produce some kind of numeric data, like an external temperature sensor, preprocessed data from external sources or elaboration based on user interactions. For the latter type of sensors, an example could be the number of interactions with the app or a ratio based on the values of other sensors.

4.1.4 Reward

The last concept is the *Reward* system, which includes an elaboration that produces a numeric value based on the different parameters and it is what defines the goal the agent has to pursue. The reward thus must be defined by the developer that declares the final purpose of its application. My package provides a template structure with which the developer can implement its reward system, it can include different types of rewards that can be exclusive among them or not. The custom reward system is a must for those packages that aim to allow the building of a custom environment.

4.1.5 Interaction

All the parts described before are created to work together and compose a *Reinforcement Learning environment*, similar to an Atari game or a simulator program as seen in the Related Work (chapter 3). The interaction is possible thanks to the engineering of all these concepts as a module of a system that is easy to develop, change and combine. The interaction is defined as how each module is built to cooperate and produce the correct behavior.

The environment is what is in charge to use every other module without knowing what they are doing. It knows only what the Actions defined are, the Sensors to be read, and the Rewards to be called. How the actions act in the real device, how the sensor produces its value, or how the reward is elaborate does not depend on the environment but needs those modules to work. *The Environment is characterized by the Sensors, the Actions and the Rewards given to it.*

The environment workflow starts at the instantiation when it takes and checks the Sensors of the pre-developed set, then saves the accepted ones into a list of observed objects. Moreover, it checks the possible external sensors given to it and adds to the same list, defining the *state* of the environment. *The State is defined as the returns of a single read of each sensor passed to the environment.*

After the reading, the information is given back to the caller.

The modality of the act in the environment involves the selection of action and then the passage of the action identifier to the environment, which then checks if the action is valid (by checking the observable list defined previously) and let it act as it is implemented. The idea is that the action is external to the environment, is it used in it and not as part of it. The necessary data for making the selection is the state, which must contain all the information for the agent. *An Action is selected by the agent with the use of the State.*

The last thing a Reinforcement Learning environment must elaborate on is the reward. It is defined by the developer and the framework does not know what are the parameters that define a reward. For the elaboration of the reward, the necessary data include the state, the action, and the next state that can be used by the developer to define its reward system. *The Reward is elaborated using the state, the action, and the next state.*

4.2 Agent

The *Agent* is the implementation of a Reinforcement Learning Algorithm. It defines the strategy followed by the agent during its interaction with the environment. The Agent is the learner: based on its moves and how the environment reacts it learns how to achieve the goal. Based on RL theory the

Agent can either know parts of the rules of the environment or does not know anything. In this project, the Agent is defined to follow the latter type, but in the chapter 7.3 we have talked about new agents that can be of the other types. The Agent is what acts in the environment, so what reads the state and chooses the action, thus it knows how the state is made and about what the actions are. In literature there are a lot of different algorithms that can be applied to an environment, in this project, there are implemented two Agents, one is Q-Learning and the other is Deep Q-Network, it's easy to implement new Agents thanks to the template provided (see chapter 7.3). The behavior of the Agent is parameterized and can be customized by the developer almost in every characteristic (see chapter 5.2).

4.3 Workflow

The implementation of the framework follows a simple workflow that is useful to easily create the final Reinforcement Learning system and let the developer focus on the application that will be used.

The instantiation part can be described as:

1. The creation of the Actions, identified by the id *i*: A_i
2. The creation of the Rewards, identified by the id *j*: R_j
3. The (optional) creation of new Sensors, identified by the id *k*: S_k
4. The (optional) selection of predefined Sensors, i will use the same notation as the new Sensors: S_k
5. The instantiation of the Environment, giving it the just defined Actions, Rewards, and Sensors, plus the parameter to customize its behavior (see Customization Section): E.
6. The instantiation of the Agent, giving it the just defined Environment, plus the parameter to customize its behavior (see 5.1): L.

After the definition of all the parts of the framework necessary to make it work properly, it is necessary to start the Agent, then the workflow goes as defined:

1. The environment E at time *t* read from each sensor S_k
2. The state s_t is produced and given to the agent L.
3. The agent read the state and select the action a_t and give it back to the environment.
4. The environment execute the action and (optionally) produce the next state s_{t+1}
5. The environment executes the Rewards R_j and produce the reward for the time instant *t*: r_t.
6. Return to the first point of this list with the time instant *t+1* as *t*.

It is possible to customize some part of this, we will see it later. The workflow is kept simple to be able to manage different applications and is strongly related to the theory of Reinforcement Learning.

4.4 Use Case Examples

The proposed framework leaves the freedom to model the environment without limit, this means that the actual opportunity of using this framework is endless and only limited by the developer's imagination.

Low Energy Mode Advice. An application of this framework could be an App including an algorithm that advises the iPhone user when the low energy mode can be activated to save battery. In these use cases, the algorithm needs to take as an observation the hours of the day (maybe divided in 30 minutes), the user's usage in the same range of time, and maybe some other sensors like the movement or something correlated to understand when the user can save battery given that he's not using the phone. The reward for the algorithm could be positive if the user accepts the suggestion and negative in case of the user is not accepting (visualizing but not activating the model). Here the only way to act is notify to the user of something, it's impossible to directly activate the low energy mode.

Health Package. Talking about the amount of data that can be reached and analyzed using the Health Kit (and relatively with the Health App) is possible to image an algorithm that suggests standing up, or taking a break, but more generally to suggest an active behavior of the user like ”do a walk, it's sunny outside”, ”take out the dog, it's the usual time”, and so on.

Torch. The possibility to use the camera module gives the possibility to use RL with images, an example could be a Torch App in which the Algorithm controls the brightness of the Led. This potentially can be used to control different lights connected to the phone, dimming the light w.r.t. the ambient light, and so on.

Sound Recognition. An example could be an app that uses RL to learn when a person sleeps and more precisely when it snores, given the capability to produce a sound that can stop the snoring.

External Control. The possibility of building new sensors and integrating them into the project offers the possibility to use the device to control external hardware. For example, a robot has to learn how to clean the house. It is possible to connect the device to the robot and let the agent command the robot and find the best way to clean every part. Some races use robots and Machine Learning to follow a line track and complete it in the least seconds possible, this project could provide a different way to approach that type of competition with RL techniques.

Chapter 5

Implementation

In this section, we will describe the implementation of the Concepts explained above. The implementation includes information about the APIs used as well as the programming logic used to pursue the goal of usability, customization, and efficiency. The Project is based on Swift, specifically the CoreML Kit and the CoreML Tools. Both have their own specific goal and are used in different ways and at different times. To build the project we faced a difficult task, mainly because we wanted to develop a framework for Reinforcement Learning with APIs built to create mobile applications. So there is a lack of basic APIs common in the ML-specific frameworks. As seen in the previous chapter the structure of the project is quite standard for an RL framework, it is composed of an Environment and an Agent that interact with each other. The performance is defined by the cumulative reward the agent can collect during a defined amount of time. *The environment and the agent begin a loop in which they share information to let the agent explore and learn.*

The applications that will use this package have to instantiate all the necessary objects before starting the learning process. To customize the environment the developer will create the object for the actions, the reward, and eventually the additional sensors and policies. In the appendix is extensively described the APIs, as well as on the documentation provided online at the URL: `https://github.com/pavva94/SwiftRLLib`.

5.1 Environment

Usually, in research, it is possible to see the implementation of a new Environment, a new game, or a new simulator with which an algorithm is trained to prove the application on that environment of Reinforcement Learning. In this project we will show an API to build environments and then apply the algorithms, the difference is important. The capabilities are a lot more.

In this project what is called *environment* is an API to create different contexts, with different properties and different rules. Usually, the Environment, defined as the object that receives the actions by the agent and executes those actions, is a completed program with its own rules and its behavior. In this project, the Environment is a customizable object, with the necessity to use custom rules and custom behavior. Though it allows creating almost infinite different environments. The environment's basic logic is the only part of its behavior that can not be modified. This involves the instantiation, the reading logic, and the elaboration of the reward system.

During the instantiation phase, it takes as input the Sensors and the Actions, as well as the Rewards objects (more on the appendix A.1). The environment needs this information to create a system that interacts with the device and the agent. The actions will be executed when selected by the agent and communicated to the environment. The actions are identified by a numeric id, of type *integer*, the agent selects an action by its id and shares it with the environment that executes the action. The execution is independent of the environment, it acts on the device according to the limits imposed by Apple (see chapter 7.2). The rewards as well are identified by a numeric id, of type *integer* (see chapter 5.3). In this phase, the environment is used as a passive object. After the instantiation the environment is ready to start, it has to be attached to an Agent to start producing data and acting on the device.

Another not editable part of the behavior is how the environment will retrieve the information when asked. It uses the shared API of the Sensor class calling all the sensors given in the instantiation phase.

The same discussion about the environment's behavior applies to the retrieving of the reward value. During this phase, the environment takes care of calling the Reward objects given in the instantiation phase. The reward class shares an API that has to returns *double* values that are then summed together to compose the final reward for the current time step. This choice was made to simplify the communication and the flow of information without loading the device excessively (see chapter 5.4).

5.2 Agent

The Agent is the class that implements the Reinforcement Learning algorithm used to explore and learn through interaction with the environment. During the instantiation, the agent has to receive the environment created previously, and additionally the policy it has to follow. It is possible to customize the agents, we will show you later the parameter. During the interaction with the environment, the agent has to take care of multiple things, including stuff to apply the learning algorithm, as well as the managing of the files.

The Agent, as opposed to the Environment, has a single implementation for each algorithm implemented. The package allows to build and integrate all kinds of algorithms, and to assure they implement the required APIs to correctly interact with the environment. The package contains a superclass that has to be inherited by the new agents that will be developed. Moreover, the superclass includes some of the methods already implemented to maintain some common parts. The *initializer*, for example, is implemented to provide a standard setting of the basic parameters. The future agents will require to set up only the path, plus custom variables. If the environment creates the states and executes the actions, the agent is in charge of learning from these states and selecting the actions to take at each time step. To do this it has to take care of the data and the training, as well as communicate its choices to the environment. To manage the data means the agents have to take care of saving the structured data that will be used to train, take care of the files in which the network is stored, and eventually save the history of its action for later use.

Sarsa tuple

A common characteristic on almost every Reinforcement Learning is the use of a data structure that includes the notion of a *SARSA* tuple. This structure takes its name from its component: the State at time *t*: S_t, the Action selected at time *t*: A_t, the numeric Reward gained at time *t+1*: R_{t+1}, the State at time *t+1*: S_{t+1} and finally the action taken at time *t+1*: A_{t+1}. All these values are used from the algorithm to update its weights as seen in the background chapter (Chapter 2). The tuples can be built easily thanks to the use of the provided APIs. The Sarsa tuple is the basic datatype with which the data are used by the agent (more on the appendix A.4).

Experience Replay Buffer

To keep track of the agent's actions and how it explores the environment we need an object that stores and provides data at the right moment. To exploit a theoretic concept we used the Experience Replay Buffer, a FIFO (First In First Out) buffer in which the data are stored to be ready for future use. The buffer is a customizable class that store and provides data as lists, and also manages the permanent storage in the device memory. The buffer is used to create the training dataset each time the Agent calls it. The procedure to prepare the data is simply the random extraction of tuples from the stored data. The size of the training dataset can be fixed by the developer, the default number is 256 tuples.

Data Manager

Given the common use of Sarsa tuples among the agents, it is possible to create a unified Data Manager to manage the data into the device to simplify the sharing of data. The data are converted into a new datatype called *Database-Data* more flexible and more easily to store into the device memory. The data manager uses this new datatype to save the information needed, it is used by the agent to store the data for the training and the history. The Data Manager will save the data into the device folder called Documents. There are other possible places in which to save the data but is the most reliable and safe because the data will remain in the app folder and will be deleted when the app is deleted (see chapter 7.1).

Parameter Customization

During the instantiating is possible to set different parameters to customize the behavior of the agent in every part. There are parameters to modify the seconds between two calls of the observation and training processes, as well as parameters to customize the hyperparameters of the training. Moreover, it also exposes some basic parameters to modify the filenames produced by the agent. The possibilities are different:

- *agentID*: used to define an identifier to the agent
- *bufferPath*: used to define a custom path for the buffer. Default based on the ID.
- *databasePath*: used to define a custom path for the database. Default based on the ID.
- *batchSize*: the size of the training batch. Fixed to 64
- *trainingSetSize*: the size of the training set to prepare. Fixed to 256.

- *learning_rate*: the learning rate used for the training, can be a list of values.
- *epochs*: the number of epochs to do in a training
- *secondsObserveProcess*: the number of seconds after which the Observe process is called.
- *secondsTrainProcess*: the number of seconds after which the Train process is called.
- *episodeEnd*: this parameter is used to pass a boolean function with which the agent checks the end of the episode

These parameters, together with the custom policy, the custom actions, and the custom rewards allow to fully customize every aspect of the Reinforcement Learning framework. An important parameter is the *episode end*. In a continuous environment like the one provided it is possible to not have the final state to end an episode. But we provide the possibility to implement control over the state to check the end of the episode through a function that returns a Boolean value. This is necessary due to the nature of RL algorithms, which have a different update rule for the final state.

MultiAgent

The structure of the package allows the use of different environments and different agents at the same time. Of course, we have to warn the reader that the use of more than one agent could compromise the device in terms of battery consumption and hardware usage, and also it can increase the heat produced by the hardware that can damage the device in extreme conditions. The possibility of using multiple agents must distinguish all the relative files and processes needed, as well as all the objects created. In regards to the files, like the model or the buffer and database, the package uses the *agentID* to define files for each one. It is possible to use different agents in the same environment, as well as different agents in different environments. The multi-agent system is developed to help the developers to find the correct hyperparameters for their problems. It doesn't allow to use of different networks, at the start each agent has the same network. Only after the first training, each agent has a different network. Of course, this is valid only with the initial network, at the first training process each agent saves its network in different files and uses it to continue the exploration. The other possibility is the use of multiple agents that interact with the same environment. They can cooperate or not, as well as perform different tasks. The developer is in charge of finding new ways to apply this feature.

To explore and learn the agent has to use two different processes that work asynchronously. These process are called *Observe process* and *Train process*. To start these processes the Agent exposes a unique method, with which to define the *Working Mode* and the *Agent Mode*. These two are arguments define what the agent has to do and how to do it. The Working Mode is dedicated to setting how the agent has to work, if in Foreground mode, in Background mode, or both together. The Agent Mode defines if the agent has to continue to learn from the data (with the customized process) or to run in inference mode, and therefore update its knowledge. This is the main and nearly the only API that the developer must use after the definition of all the objects (actions, rewards, eventually sensors), and after setting the parameters (more on the appendix A.2).

5.2.1 Observe Process

The process of reading and collecting the state, as well as selecting the action and calling the environment is named *Observe* process. It is managed by the agent through two different scenarios the developer can choose:

- *Foreground observation*: the process happens after a defined number of seconds, using a Timer.

- *Background observation*: the process takes place through the Background Task APIs exposed by Apple.

The difference between the two types is disparate. The Foreground observation, as the name suggests, works only when the application is used in the foreground during the usage. It is a method that uses the Timer class to execute the Observe process, it is reliable as far as the punctuality of execution is concerned, but has the limit of being unpractical if the App needs to work also in standby mode.

The Background observation works in every condition, both if the app is in the foreground and suspended in the background. When the app is in the background the process takes place in a different thread w.r.t. the application but there is almost no limit to what it can do. This method uses the Background Task APIs by Apple that has a complex workflow. When the background process is launched it goes into a queue managed by the Apple operating system which has its own rules. The Apple background manager concretely executes the task at its discretion, based on its idea of how much the application is used, the time when the user usually opens that app, and on other parameters developed internally by Apple. Only one parameter can be fixed by the developer and it is the number of seconds, from the launch of the task to its concrete execution.

These two methods to execute the Observe process can be combined to provide a constant and reliable way to apply the Reinforcement Learning flow of time. The choice of which way to use is delegated to the developer, to leave the possibility of customizing the behavior of the environment open.

When the developer chooses what modes prefer and start it, the operational flow goes as described in the Workflow section (chapter 4.3), though the agent calls the environment that gives back the state, then the agent selects and communicates the decision to the environment that executes it and returns the reward. During this process the agent has to save the data to be used in the Train process, using the DataManager.

5.2.2 Train Process

The *Train* process is the other main part of the agent behaviors, it applies the Reinforcement Learning algorithms to train the agent. It is executed using the same idea of the observed process, using the two types of execution:

- *Foreground training*: the training is made only when the app is in the foreground
- *Background training*: the process takes place in the background

The training can take place by default only when the device is on-charge to not affect the battery and so the user's experience. This choice was made to restrict the possibility of causing problems to the device through a wrong configuration of the training parameter as well as the use of over-the-limit computational power (see Chapter 7.2). As for the Observe process, it is necessary to define the number of seconds to wait until the task can be executed, with the same limits as the Observe process. There is a difference between the background Observe process and this one, the different types of tasks from the point of view of Swift. In the case of the observation process, it includes the execution of reading the state, determining the action to take, and calling the environment to continue the process. It is a process that takes little time to be completed. Given this property, it is possible to use the BGAppRefreshTask class of Swift. This class can be used to execute some kind of update of the app that at most takes 30 seconds. Regarding the background training process, instead, the property of a lightweight execution can not be assumed at all as the data have to be prepared and Swift exposes the BGProcessingTask class. This class allows executing code without a time limit.

The Training process is divided into two parts, the first one creates the training data starting from the Sarsa tuples saved. The data retrieved by the buffer needs to be modified to be effectively used in the train. It must be structured into the input and the target. The target is calculated based on the RL algorithms used, so each Agent has its way to define the target according to the Sarsa tuples. The common workflow includes the generic update management, with the use of mini-batch to update the learner, taken from a training dataset created as described before.

The training process uses the data produced by the observe process and retrieved with the use of the Buffer and the DataManager, and then elaborated by the Agent. If the Observe process is common to all the algorithms implemented, the training process has to be specifically developed for each algorithm.

5.2.3 Algorithms

The framework supports two Reinforcement Learning algorithms. The *Q-Learning* and the *Deep Q-Learning*, the last one implemented with the use of a neural network.

Q-Learning

The *Q-Learning* class is built to implement the homonym algorithm. Q-Learning works with environments that have a defined number of states and actions. The class though needs to know what are the number of the state and the actions to create its representation, in this case, implemented with a bi-dimensional matrix. The matrix includes a value for each state-action pair that represents the value of the estimated future reward. The update of these values follows the formula described in Chapter 2.2.6. The matrix starts empty, the agent doesn't know the state, during the interaction with the environment the agent checks if the state is already known and chose the action, otherwise it extends its matrix to include the new state and assign to it a random action value from which it chooses the action.

The training process using a matrix of double is made manually without the use of any pre-built framework. It uses the Q-Learning formula to update each weight of the matrix and makes the agent choices change.

Deep Q-Network

The *Deep Q-Network* class implements the Deep Q-Learning algorithm with the use of a neural network. The neural network is created and used thanks to the CoreML kit provided by Apple that takes care of creating a model usable into the device and exposes the CoreML APIs used to call the model. The input for the network is composed of the vector produced by the environment after the observation of the sensors. The output of the network is a vector of probabilities with the size of the number of actions. The matrix of the Q-Learning is replaced by the weights of the neural network, but the update follows the same formula. The DQN provided uses a second network used to create the target each time the training process starts (see chapter 2.2.6).

Here is the algorithm:

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

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $epsilon > 0$
Initialize $Q(s, a; \theta)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
Initialize $Q(s, a; \omega)$ equal to $Q(s, a; \theta)$
foreach episode do
    Initialize S
    foreach step of episode do
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon - greedy$)
        Take action $A$, observe $R$, $S'$
        $Q(S_t, A_t; \theta) \leftarrow Q(S_t, A_t; \theta) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \omega) - Q(S_t, A_t; \theta)]$    ▷ if $S_t$ is terminal then $max_a Q(S_{t+1}, a; \omega) = 0$
        $S \leftarrow S_{t+1}$
    end foreach
end foreach
```

The agent has to take care of the network, in terms of managing the files in which the network is saved. This is because CoreML uses the URL of the network to keep track of it. A motivation for this can be that keeping in memory the model with its weights is too heavy in terms of computational power. Furthermore, the loading and the saving process are optimized and they do not consume much power. A peculiarity of the application developed for the Apple ecosystem is that the network created and added to a project is transformed into a class that inherits from the official superclass MLModel, thus all the methods are inherited and the network becomes a callable class. A limit to this process is that the name of the network, and so the class attached to it, must not change during the whole development. We fixed the name used in the package, the same that must be used in every app, to *RLModel*.

The training is done with the CoreML function *MLUpdateTask*. It takes care of the model training given the model URL, the training data, the training configuration, and two handlers to call during the training progress and when the task is finished. The training configuration includes all the most common model parameters used during training:

- *Learning Rate*: the learning rate used for the update rule
- *Epochs*: the number of epochs for each training
- *Mini batch size*: parameter used to specify the size of a miniBatch used by optimizer
- *Epsilon*: parameter used to control the epsilon of Adam optimizer
- *Beta*: parameter used to control the beta of Adam optimizer
- *Momentum*: parameter used to control the momentum of gradient based optimizers
- *Shuffle*: parameter used to specify whether to shuffle the data between epochs

These are the keys used to configure the training, with which a dictionary is created and the desired values are assigned.

5.3 Action

The actions have to be built by the developer following the protocol provided by the framework. They need to be identified by a numeric ID, of datatype *integer* and have to implement some defined functions to be correctly used by the framework. After the instantiation, the actions have to be passed to the environment to be called when selected. The protocol is defined as:

- *ID*: the identifier of the action
- *Description*: a text description of the actions
- *exec*: the function called by the environment when the action is selected for execution

The actions are created by the developer, who uses this object to implement the interaction with the real device. It can do anything in these objects, the developer is free to choose how to act, if an action has to modify some device property or if it has to notify the user or command something external (more on the appendix A.3.1).

5.4 Reward

The reward system of the environment is based on the Reward object set by the developer and given to the environment at the instantiation phase. A reward has to implements the whole logic to evaluate the current state, the action chosen and the next state to produce a single *double* value. The protocol is defined as:

- *ID*: the identifier of the reward

- *Description*: a text description of the reward
- *exec*: the function called by the environment when the reward is needed

The rewards created by the developer will define the goals of the agent, they are free to develop every idea they had (more on the appendix A.3.2).

5.5 Sensor

To leave the possibility of extending the capability of the framework it is possible to create new sensors, with which is possible to create new ways of retrieving the data that will build the State. It is provided an example class to create new sensors with the correct structure. The output of the sensor is RLStateType, as defined before, it does not limit the usage possibilities (more on the appendix A.4).

It is possible to ask the environment to give the sensor the data of the other sensors, this feature can help to build sensors that use the environment data to elaborate a new value.

To standardize the structure of this class it is provided a protocol, called *observableData*, from which every sensor inherits. The protocol is defined as:

- *name*: a text that identifies the sensor
- *stateSize*: the size of the output
- *read*: the function called by the environment when the sensors have to be read
- *preprocessing*: a function where to include the preprocessing

The library includes different Sensors, already implemented. Those Sensors include some of the basic updatable parameters a device has, as well as normal information about the clock or the date. As said the sensors can be anything and can be easily built to manage every hardware or every piece of data.

The pre-developed Sensors are:

- *Altitude*: reads the actual altitude of the device
- *Battery*: reads the battery value
- *Brightness*: reads the screen brightness
- *Barometer*: reads the current pressure
- *Clock*: returns the hour, minute, and seconds
- *Date*: returns the current date divided in year, month, and day
- *Gyroscope*: returns the X, Y, Z values taken from the Gyroscope
- *Hour*: returns only the current hour
- *Location*: returns the coordinate of the device
- *Locked*: reads if the device is locked or not
- *LowPowerMode*: checks if the Low Power Mode is active
- *Minute*: returns only the current minutes
- *Orientation*: read the screen orientation
- *Second*: returns only the current seconds
- *Speed*: returns the speed at which the device travels
- *Volume*: returns the volume of the device

The Sensors can be used to customize the environment (more on the appendix A.3.4). It is possible to implement sensors that take in the values external to the device. For example, if the device is connected to external hardware it is possible to retrieve information from it, and maybe use that information to act on this hardware. This means using the device only for the computation. It is possible to train an RL algorithm in an Apple device only by creating the correct sensors.

5.6 Policy

The policy is another editable object. The package provides an implementation of the Epsilon Greedy policy and a protocol to be used to implements other policies. The policy can be used by different agents, as well as the previous concepts, but it is a bit more complicated because it works with the model. To use the model the developer needs to know the basics of the CoreML kit. We exposed an API to call the model given an input, only for the policy objects.

The protocol is defined as:

- *ID*: the identifier of the reward
- *Description*: a text description of the reward
- *exec*: the function called by the environment when the policy is needed

The most common policy used is the Epsilon Greedy, presented in the Background section. To help the future developer we implemented it.

5.7 Datatype

I used a special feature of Swift, we used a *typealias* for the common datatype used in all the package. We decided to use a *list of Double* as the type used in the package to carry information to all the components. The choice was made to keep the framework simple, without limiting the customization. Almost all the data types can be turned into a list of double or in a single double. It is possible to think that the increase of the memory footprint using double instead of float is significant but the power provided by the current generation of smartphones allows one to use it without any problems. Moreover to standardize the datatypes in the project we created two type alias for the *Integer* and the *Double*.

```
// The alias for the datatype used by the observableData
public typealias RLStateType = [Double]
// The alias for the datatype used by the action
public typealias RLActionType = Int
// The alias for the datatype used by the reward
public typealias RLRewardType = Double
```

5.8 Use Case

A package alone can be evaluated using the metrics above or by software engineer experts to check the correct use of patterns, basic code rules, other than the usability or the applicability. All these evaluation phases are important but the most important question about a package is if it works. To answer this question, and to explain what is possible to build with this package, we developed some applications to show the potentiality and to evaluate the performance of a Reinforcement Learning environment on mobile devices.

5.8.1 Battery Manager

If we wonder about what Artificial Intelligence can do to help me during the day, it comes to me that it can remind me of my appointment, or show me some news that interests me. These are tasks that can be done with my mobile device, nowadays, a smartphone is one of the most used things because can assist the user in his daily life. However one of the struggles with smartphones is the battery drain during the day. The brands of smartphones, or more broadly of all devices with a battery, already use algorithms to minimize battery consumption. These algorithms can be based on Artificial Intelligence, used to find usage patterns and act accordingly giving more power when necessary or stopping some tasks when not explicitly requested. They could use Machine Learning techniques to find patterns or to find other ways to limit battery consumption. This means using a lot of data collected from all possible devices and applying the model to every single device without differences. If it is possible to create a model that studies the behavior of every single user and tries to minimize battery consumption, it could be the highest level of personalization on a device. Reinforcement Learning and this package provide the functionality to create an RL environment to manage the battery and train with data collected only from the interaction between the device and the agent. Some limits stands, for example, Apple does not allow to modify the phone settings without the user intervention outside the application in the foreground: the agent can act directly only when the application is active in the foreground. This is a limits given by the Apple Ecosystem, but for the sake of this study, it is possible to ignore the fact that an application will never always be in the foreground.

To create an environment with which the agent can learn how to decrease battery consumption we used the sensors built in the package. To simplify the work of the agent we selected only some sensors: *BatterySensor*, *BrightnessSensor*, *Clock*. We decided to limit the agent's actions to the possibility of modifying the brightness of the screen, one of the most energy consumption hardware on a device. The actions are: *Increase the brightness* and *Decrease the brightness*. It is possible to extend this example with more actions on different devices, or with more action for the same parameter (e.g. declare a new action to do not modify the brightness). Regarding the time step, we chose to let the agent act once every half an hour. This example can show how the Reinforcement Learning agent can learn the user's behavior and decrease the brightness when possible, we left the action to increase the brightness to let the agent learn when the user wants to see the screen. In this example, the reward is based on how much the battery lasts during the day. We propose two different reward systems I've tried. The first one gives the agent a positive reward for each step in which the battery is not zero, zero otherwise. This reward encourages the agent to increase the number of steps, and so to make the battery last longer. The second reward system is based on decreasing the consumption w.r.t. the previous step. A positive reward is given to the agent when it gets better (to read as lower) consumption, and zero if the consumption stays the same or increases. For this example we had to set a state as the final state for an episode, I've implemented a function that if the battery value reaches zero that is the episode end, will be used inside the agent when the training data are created.

5.8.2 Notification Manager

The customization property of this package allows the creation of disparate actions, from modifying the settings to interacting with the user. The latter case is the second example app we developed: an application that has the purpose to send notifications based on when the user usually reads them. This is a use case that can be useful to all the applications that have to notify something not urgent to the user. For example, this technique can be used for advertisements, to have a higher interaction with the user, sending a notification with the advertisement only when the user would usually read it. For this example we chose to build the Environment with different sensors w.r.t. the previous use case: *Locked*, *Battery*, *Clock*, *LowPowerMode*. Those sensors have been chosen because with the information they provide the agent can understand when the user will be active and read the notification. In this usage case the actions are: *Send* the notification or *Not Send* the notification. To connect what the agent does with the state it reads it is necessary to create a new sensor that includes an algorithm to keep track of the notification. To build this new sensor we devised a way to keep track of each time step, so each half an hour, with the history of the last 5 notifications. This information is based both on the user's and the agent's behavior and connects them easily. This new sensor is added to the environment with the correspondent API. The reward is based on the new sensor, if the ratio calculated by summing the read notifications and the sent notifications is higher than the ratio of the previous step (at the same timestep) then it takes a positive reward.

5.8.3 Environment 1D

The use of the Q-Learning agent is limited, due to its property of using a matrix to keep track of the knowledge acquired. This can bring to higher use of the memory as well as fill the bandwidth of the device is used for problems with a huge number of states or actions. So to prove the working of this specific agent we built a new simple environment similar to those seen in the basic literature. We am talking about environments that can be visualized through a matrix or a vector. In this case, we built a vector environment to reach the position with the positive reward and accumulate the maximum reward possible in a single episode. There is a position in which the agent takes a negative reward and it needs to learn to pass over it and reach the state with the maximum reward. This use case is interesting to show that the presented framework can be used also without any information about the device. It can be used to simulate an RL environment and run it. This is an example that shows the potentiality of my package, it allows me to use the device as a training machine and not only as an environment itself. The reward system is designed like in the simplest Reinforcement Learning examples, when the agent reaches the rightmost position it takes a positive reward. In the middle, there is a position that gives a negative reward. The agent always starts at the leftmost position.

Chapter 6

Results

When a new framework is developed it can be difficult to evaluate the results, especially if the framework is the first of its kind. The package described in this study is the first Reinforcement Learning framework for the Apple ecosystem, so we will evaluate it with some statistics about the code and then assess the performance by comparing different architectures, different hyperparameters, and more. The tests are done in the device simulator provided by Xcode, as well as in a real device that is the iPhone Xs Max. The use of the Xcode simulator simplifies the testing without a real device, especially for the initial development part. The Xcode simulator is only an emulator for the interface, it does not simulate the hardware, which can not be done in an Intel Mac given the ARM processor that Apple devices have. It is built to help the developer with the testing of their app's User Interface but not much more. The limit of some simulators is that the hardware is not virtualized, so the settings are not coherent with the real device, though it was necessary to develop a Simulator for the hardware part, which is fundamental for testing the applications correctly. Human behavior can be simulated as well using the simulator developed.

6.1 Code Metrics

As far as the code is concerned, it is possible to evaluate its goodness with some metrics. The metrics chosen will describe how the framework was developed as well as show the amount of work done to complete it.

The metrics are provided by a GitHub project: Swift Code Metrics [5] and provide an interesting insight into the architectural state of the software written in Swift, consisting of several modules. Based on what is described in the book Clean Architecture by Robert C. Martin [15], the package expose:

- *LOC*: Lines Of Code
- *NOC*: Numbers Of Comments
- *POC*: Percentage Of Comments
- *NOM*: Number of Methods
- *Number of concretes*: Number of classes and structs

These metrics show what the architecture of the package is like. They are important to understand if the package is well written and allows to analyze what has been done to simplify the life of future developers. For example, the Line Of Code can be used to compare the work done with the work the future developer will do using my framework.

The results for these metrics are:

Metric	**Value**
LOC	1775
NOC	711
POC	29%
NOM	119
Number of concretes	44

Table 6.1: Code metrics.

The first metric, the *LOC*, can be used to compare the number of lines written in the package with the lines of code necessary to create an RL environment and an Agent using my package. From 1775 lines of code the final number for the developer becomes less than 50, but it is based on how customized is the application the developer needs. If the developer uses the basic hyperparameter and the epsilongreedy policy provided, it is possible to create only the actions and the rewards. The use of my package allows starting an RL environment into a mobile device with a very little fraction of the lines of code necessary to create the framework from scratch.

The *NOC* describes if the code is well commented from a developer point of view, a very important feature of the software, and in this case, it has the 29% of POC which can be considered a good number of comments if we analyze the study by Oliver Arafat and Dirk Riehle [2]. The *number of methods* is 119, the great use of different methods is involved in the Software Engineer mechanism of writing code. Usually, it is a good practice to isolate the specific functionality inside a single method, that will be used only for a specific task. The same reasoning can be done with the *number of concretes*, in this project there are 44 classes, equally divided into the different parts of the project, as it is possible to see in the figure 6.1:

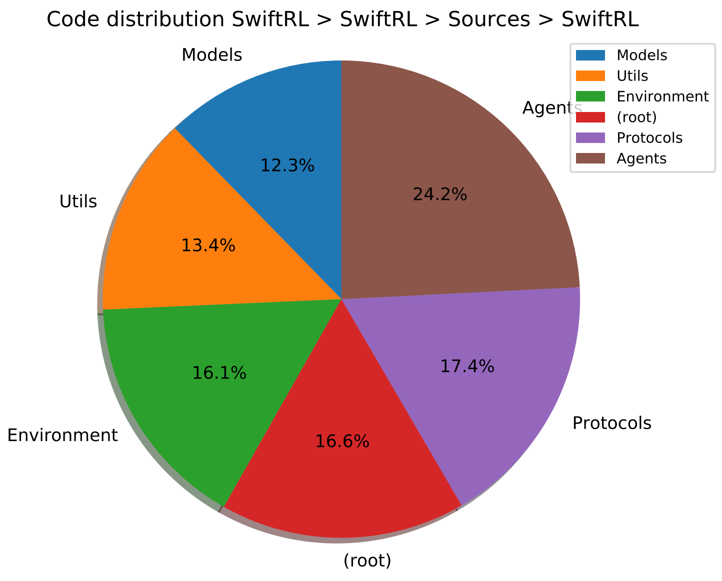


Figure 6.1: Code Distribution of Swift-RL Lib.

6.2 Device Metrics

In this section, we will evaluate the parameter regarding the device, for example, the CPU used during the Observe process or the Train process, as well as the memory use and the energy footprint. All these parameters depend on the applications developed.

6.2.1 Processor

The use of the CPU and the GPU, as well as the Neural Engine when available, is one of the parameters that can influence the usability of the package from a developer's point of view and is a great source of energy consumption. To evaluate the processor consumption we used the only real iPhone we got, the iPhone Xs Max that also has the Neural Engine. The use of the Neural Engine is not under our control, only the operating system can decide if the application can use it and when it can, so it is difficult to evaluate its performance.

The CPU is used for all tasks, it involves the use of the agent and its two main processes, as well as the physical training made using the CoreML kit. Let's divide the analysis into two parts, evaluating the impact of the two processes.

Observe process

The Observe process has the task of reading the Sensors at determined times, but each time the task is equal so it is possible to pick a single observe process to evaluate the whole process. The amount of processor used belongs to the number of sensors the environment has to read, from the agent's decisional part and finally to the developer control, indeed the Sensors and the Actions are developed by the developer so their performance and their processor use depend on what the developer wrote.

The test is done with 5 sensors, all belonging to the sensors provided by the package. An interesting thing is that each sensor, depending on what it does, hits the processor differently, for example reading the brightness value is straightforward while reading the location means using the GPS board and it takes a lot of CPU to do that. Another important parameter for the observation process is the number of Actions the agent can select. In this case, the actions are three. The DQN has 4 layers with a maximum dimension of 32 units, a small network.

It is necessary to distinguish between the DQN and the Q-Learning, the first uses a neural network to make the decisions, while the second uses a matrix, so the differences in how the processor is used are intrinsic to the different methods of choosing an action.

Agent	Processor Consumption
Q-Learning	$\sim 5\%$
DQN	$\sim 40\%$

Table 6.2: Average processor usage of the two agent during the observation.

The data average the daily usage. The Observe process is almost stable regarding the processor usage. The values of the table 6.2 have to be put in the range of a single thread, so between 0% and 100% but the device can use at most 8 threads, to reach a total percentage of 800%, this to show that the processor is not much used. Another interesting analysis is the CPU time used by the Observe process. Also in this case the values are influenced by the number of sensors to read and the different implementation of the agents.

Agent	Processor Time
Q-Learning	$\sim 0.2\%$
DQN	$\sim 0.5 Sec$

Table 6.3: Average processor time usage of the two agent during the observation.

The time used for each observation step is almost constant w.r.t. the time used and also the processor usage percentage. The values are calculated by the daily means over the processor time for the Observe process only.

The CPU usage in the table 6.3 for the Q-Learning agent is different in the milliseconds used, the time is lower compared with the DQN agent. It seems instantaneous, and we can say it is instantaneous because the Q-Learning uses a matrix to choose the action so it is only a value read and this example uses a very small matrix, in case of bigger matrix the time will increase proportionally.

Training Process

The training process is what consumes the processor the most. Regarding the DQN, the training process uses the CoreML APIs to train the network and it's known that the back-propagation needed is very heavy in terms of the necessary computational power. The same problem arises when trying to explore the processor usage compared to the Observe process: the developer is allowed to personalize the network so is difficult to evaluate all the possibilities.

Talking about processor consumption, the differences between the two types of agents are important, the DQN uses more the processor thanks to the use of the Apple APIs that optimizes the elaboration. The Q-Learning is made by me, so the management is manual and the processor consumption is less, thanks also to the different update processes. The training is done with the same 256, with batch_size at 32. In the table 6.4 it is shown the different CPU usage percentage.

Agent	Processor Consumption
Q-Learning	$\sim 40\%$
DQN	$\sim 90\%$

Table 6.4: Average processor usage of the two agent during the training.

Regarding the processor time, the difference is totally in the update process and not in the common process of target creation.

Agent	Processor Time
Q-Learning	$\sim 0.2 Sec$
DQN	$\sim 0.5 Sec$

Table 6.5: Average processor time usage of the two agent during the training.

The CPU usage for the training process, which include the creation of the feature and the update process, is divided in two parts also concerning the amount of CPU used. Firstly the feature creator that uses the CPU more randomly. On the other hand, the real update process of the network is more stable, with a higher CPU usage. If the training size is fixed bigger than the 256 by default the first part increases linearly w.r.t. time. The same can be said about the update process, influenced by the training size and the network size the CPU usage remains nearly the same, while the time increases linearly.

In the table 6.5 for what regards the Q-Learning the training process is smaller in terms of processor use, it has to update the matrix with the new calculated values. To do that, the main thing used is the memory buffer that is used to retrieve the matrix and save it after the update.

6.2.2 Memory

The DQN uses a network and its dimension and the training hyperparameter will go to influence the device usage. The Q-Learning also uses a matrix that can consume a lot of memory both for keeping the data and for the read/write processes.

The DQN is optimized to work with huge data and a huge network, thanks to the Apple APIs it is efficient in terms of memory bandwidth. Regarding the Q-Learning algorithm the memory bandwidth usage is high, this is caused by the implementation of the matrix, and the continuous save and load of the data to avoid losing information. The memory used is directly correlated to the network dimensions and the sensors used. The network can include at most 30 layers but their size can be arbitrary, this means the model can be really large and occupy a lot of space. The data are stored in the application folder and can grow indefinitely with the training, but in a real use case, it is not necessary to save all the data but only the buffer file to train, which is very limited. Moreover, the fact that the package uses only Double values limits the use of the device storage. Regarding Q-Learning, it is provided to solve simple problems, so to use it within an environment with a large number of states could be problematic also in terms of memory used and bandwidth. We suggest using the Q-Learning only for static examples, in which the number of states is predefined and in a low number.

6.2.3 Energy

Energy consumption is also an important, if not the most important, parameter to check. Deploying a package that could be used to create applications for mobile devices, or at least that can have a battery, the package workflow must use the least possible amount of energy in every condition. The energy used by the Observe process is nearly the same for both the agents. The bigger difference is during the Train process, because of the intrinsic differences of the two agents, as already described. The energy consumption is correlated only to the size of the network and to what the sensors and the actions can do. But only the parts defined by the developer. It can choose what the network is like, and this implies the energy consumed to train varies according to the network size. The actions as well are defined by the developer and if an action has to calculate some complex number the energy spent is derived from it. The same can be said about the sensors, based on what they must elaborate they use more or less energy.

The energy consumption is not a problem derived from the package but correlated to what the developers will do with it.

6.3 Use Case Evaluation

The evaluation part based on the use cases presented above has to assure the performance and the correct functionality of the package developed. We will make different tests, using all the agents provided, the use cases, and then analyze the performance of the same agent with different configurations. To test the consumption in terms of power, memory, and energy we used a real device, which used real mobile hardware.

The first test is to compare an RL Agent with a Random Agent and analyze how the knowledge of the agent increases w.r.t. an agent without any learning brain. This test is done with the Battery Manager use cases. Moreover, we will compare the two Agents implemented in the same environment, to check the correct functionality. This test is made to compare how the two agents will learn, based on different paradigms. The results could be different and not comparable but it could be an interesting view of how they learn differently.

Another test is done using a different Hyperparameter for the Deep Q-Network to evaluate the changes and to find the best ones w.r.t. the environment it is applied to.

Lastly, we will compare the same agent with different networks, with different sizes, to understand the capabilities of the device to manage different networks and to analyze the learning differences on the same problem.

To make all those tests we developed a simulator to help me accelerate the process.

6.3.1 Simulator

During the testing of the app, we have to simulate the user's behavior during the day, as well as the device settings and property (e.g. the battery consumption). To simplify and speed up the testing we built a Simulator. The Simulator is a class that simulates some of the main characteristics of the device, for example, the battery consumption or the screen brightness. Of course, it is impossible to simulate all the settings of the device, we focused on those setting we need in the use cases. The simulation is not realistic in terms of values produced but it can simulate much more easily the device with which to test the package performance. The main simulated setting is the battery consumption, it is calculated through the use of static consumption for each hardware the agent can control. To make myself clearer, we used a base consumption of 2% for each timestep to simulate the continuous battery discharge caused by the device itself. Additionally, each hardware attached to it, for example, the use of the Wi-Fi board or the Bluetooth activation, increases the consumption with different percentages. we included only the settings we will use for the use cases, plus some others, like the low power mode and the Wi-Fi and Bluetooth boards.

- *Base Consumption*: the device itself uses energy to stay active, so it uses 2% at each step
- *Screen Brightness*: the consumption ranges from 0% to 10%
- *Wi-Fi*: if activated it consumes 3% more
- *Bluetooth*: if activated it consumes 2% more
- *Low Power Mode*: if activated it consumes 1.5% less

Another important simulated property is the Clock to speed up the interaction it is necessary to simulate a lot of days in the shortest time possible. To do that we implemented a simple fake clock that produces values for each half an hour. Each use of the Simulator includes the generation of fake times to define the timesteps. Having this simulator is a huge game-changer to produce a great quantity of training data, it is crucial to train the agent as fast as possible. I've integrated the simulator into the environment to be used also in a multi-agent context.

6.3.2 Battery Manager

Regarding the Battery Manager use case, the settings to simulate are the battery consumption and the screen brightness. They are the settings the agent can modify with the actions to try to limit the battery consumption. It is possible to increase the complexity and so the capability of the agent by adding Wi-Fi and Bluetooth management. The simulator is used not only to simulate the physical hardware but also the behavior of a user. This is important because the agent needs to learn the user's habits to correctly modify the settings without generating unwanted behaviors that can compromise the interaction of the user with the application. During the test we fixed a simple user behavior to explore how the agent can realize when it can not execute some actions, for example, we fixed that the screen brightness could not be under 50% for the hours ranges between 6 pm and 9 pm, and if the agent decreases it, it would take a negative reward. The state produced by the environment includes the battery value, the screen brightness, and the hour and minutes. With these data, the agent has to learn how to decrease the consumption of the battery and let the battery last longer. The use case is described in all its parts in the relative chapter 5.8.1. The network used has 6 fully-connected layers, with at most 64 units. The agent uses an EpsilonGreedy policy in which the epsilon decreases after a pre-defined number of steps. Initially, it uses an epsilon of 0.7 for the first 5000 steps, then decreases to 0.5, and after 10000 steps it becomes 0.2. The hyperparameters are:

- *learning rate*: 0.000001
- *epochs*: 10
- *batch size*: 64
- *gamma*: 0.999

Maximise-step Reward

The first reward is the one used to maximize the step when the battery is still alive. This reward is conceptually similar to the literature example of a robot in an environment that tries to stay alive as long as it can.

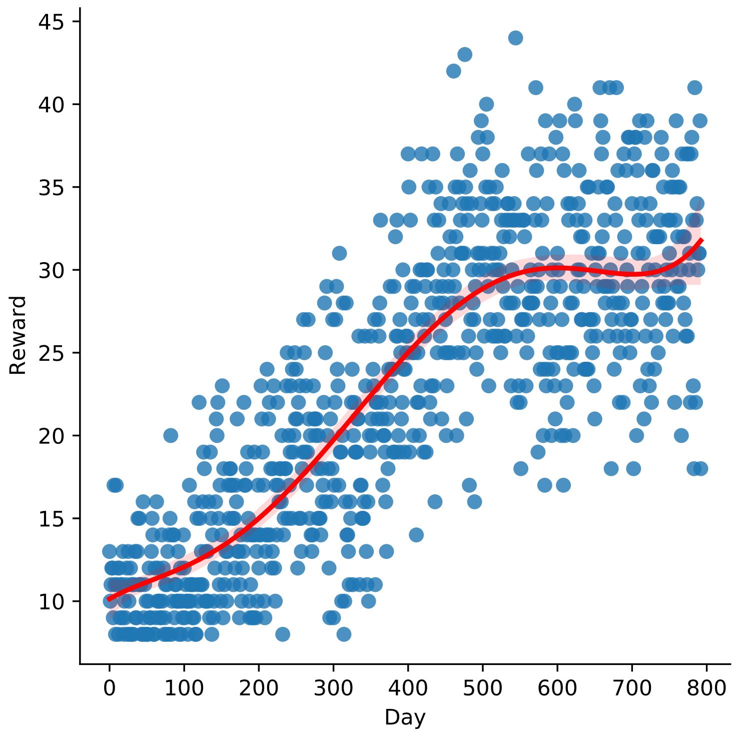


Figure 6.2: Daily reward of the agent during the training using the Maximise-step reward.

The agent starts very fast to decrease the brightness as soon as possible, and so to receive a higher cumulative reward at the end of the day. The graph shows exactly the rewards (or steps) reached summing all the daily rewards. Thanks to the figure 6.2 it is clear that it learns, it starts from 10 steps, and reaches the maximum step allowed at 45 one time. Stabilize the reward to 30 after only 600 days of simulation.

It is possible to analyze how the agent behaves differently and continuously learns the best action for each state. The graph 6.3 it is shown the comparison between the times each action is selected during the training.

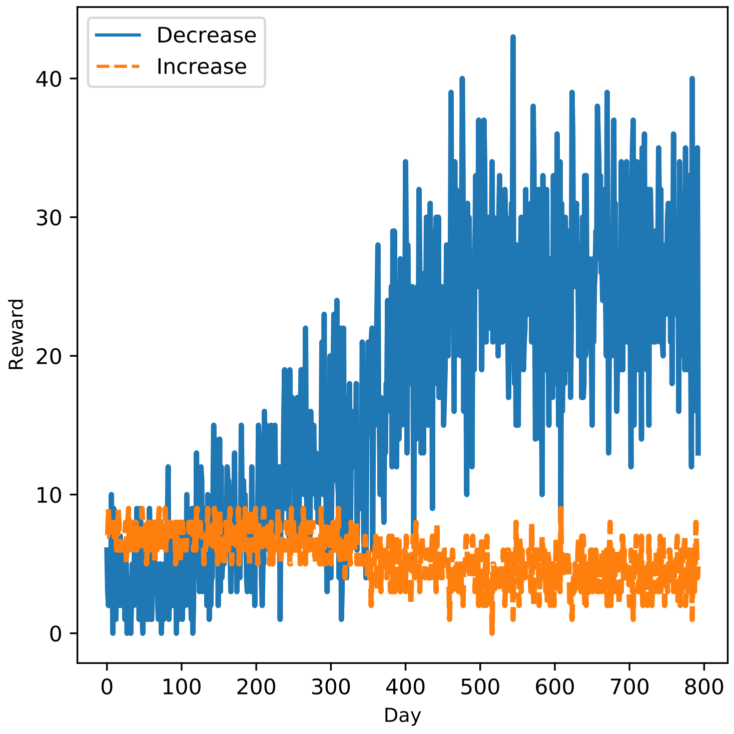


Figure 6.3: Daily actions of the agent during the training using the Maximise-step reward. Divided in Decrease and Increase actions.

Minimize Consumption

The second reward we have implemented is about the minimization of the step consumption w.r.t. the previous step consumption. In this case, the agent has different information about the optimality of the action at each step, it has to focus on the step-by-step consumption to decrease it during the time. It is a bit different from the previous one but the final behavior is the same, to make the battery last longer. The settings are the same as in the previous example.

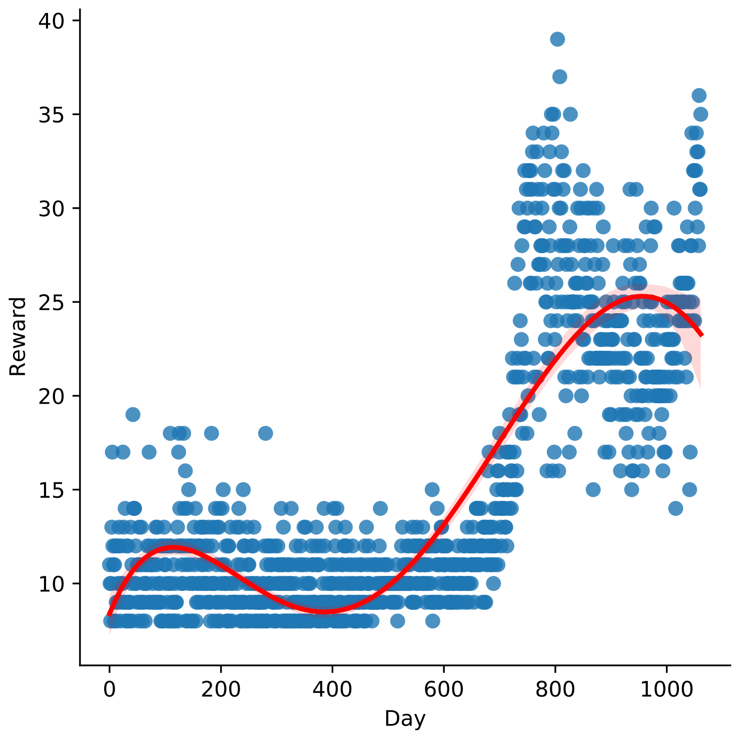


Figure 6.4: Daily reward of the agent during the training using the Maximise-step reward.

The agent uses more time to learn something useful, the problem is more complex. The red line in figure 6.4 shows the learning trend which reaches a good performance but has some final decreasing performance. It can not reach the same cumulative reward as with the previous reward but it is not so far. The action plot 6.5 is correlated with the reward and in fact, the agent starts to increase the reward when the Decrease action becomes the most chosen. This graph is really interesting in showing how much the agent learns: without the increasing of the correct action choice, the agent would continue to not increase the daily reward.

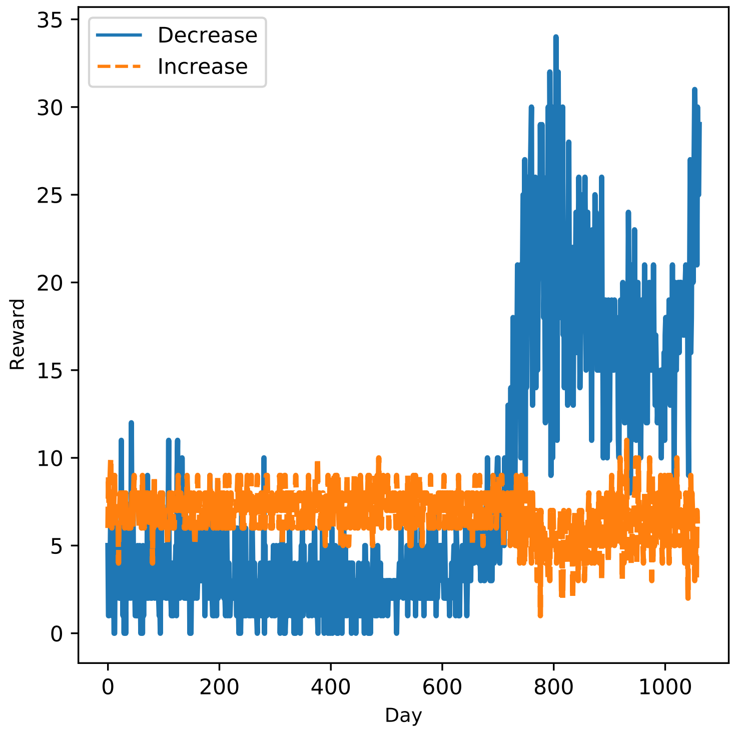


Figure 6.5: Daily actions of the agent during the training using the Minimize-consumption reward. Divided in Decrease and Increase actions.

6.3.3 Notification Manager

The Notification Manager use case uses more sensors but the actions do not modify the settings, so the simulator for this use case is limited to simulate the flow of time and the base battery consumption. In this use case, the user's behavior is what the agent studies and from which it learns when it is better to send a notification that will be read. The user's behavior is simulated using the Simulator by fixing defined hours in which the user will tap the notification a lot, and leave the other parts of the day without interactions. The agent has to learn when the user interacts the most and so when to send to the user notifications.

The network used is a simple DQN with 4 fully-connected layers of different sizes, but at most with 128 units. The input is made of 6 values defined by the sensors (see chapter 5.8.2).

To test the performance we created the graph of the reward gained by the agent during the training. The training was done with the Epsilon Greedy policy with an epsilon initially set to 0.6, which means the agents Explore for the 60% of the time, and Exploit its knowledge for the other 40% of the time. After 10000 steps the epsilon decreases to 0.3.

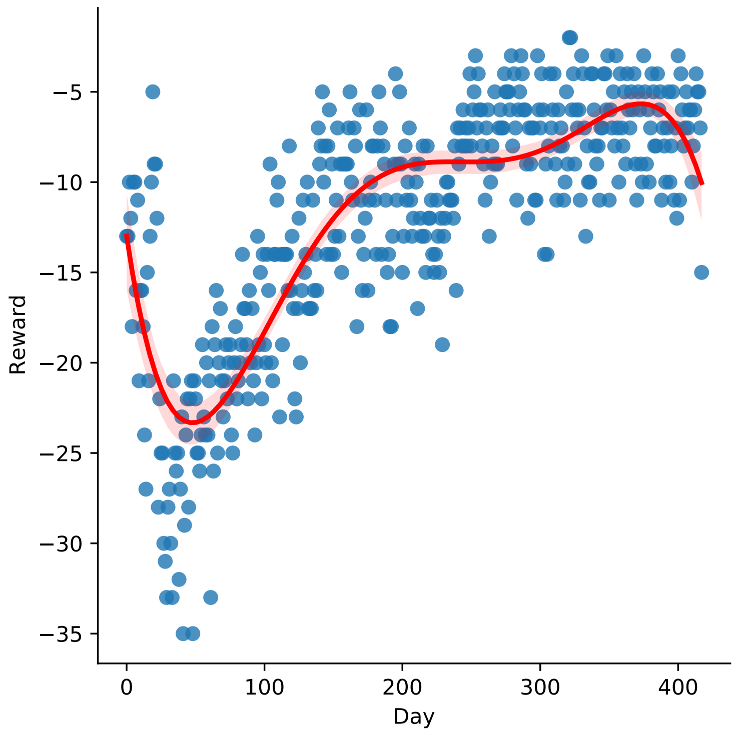


Figure 6.6: Daily rewards during training with the DQN agent.

With the help of the plot 6.6, it is possible to visualize the improvement of the agent's understanding of the user's behavior, increasing the cumulative reward of the day a lot. The graph has on the horizontal axis the days, after circa 30 days the agent uses a different epsilon to exploit its knowledge. It is clear the improvement at that point, the reward increase until it touches the 0, and goes on with the days the reward tries to increase more, reaching in some cases a positive reward. The stall in the graphs derives from the not-so-good fine-tune of the hyperparameters. Changing the epsilon of the policy after some time step it is possible to see the differences between the greedy action and the random action effect over the final reward.

It is interesting to point out the not-so-bad performance at the start, this is due to the initialization of the network's weight that fortunately produces good behavior. After the first days, the performance is getting worse, due to the exploration of the environment case by the high epsilon fixed in the Epsilon Greedy policy. During the training, the agent reaches a very good performance The fall after the high is probably due to the change in the epsilon that leads the agent towards more Exploitation. After the fall, the agent goes back to higher rewards, thanks to an improvement of its Exploitation knowledge.

The number of rewards the agent gains is directly correlated to the number of notifications it sends. Below is shown the number of notifications sent: it is correlated to the previous graph. The graph 6.7 shows that at the start, with a lot of exploration, the agent sent a high number of notifications, while to collect more rewards it has to send fewer of them but at the correct times. That is, the agent at the highest point of the reward graphs sent the lowest number of notifications.

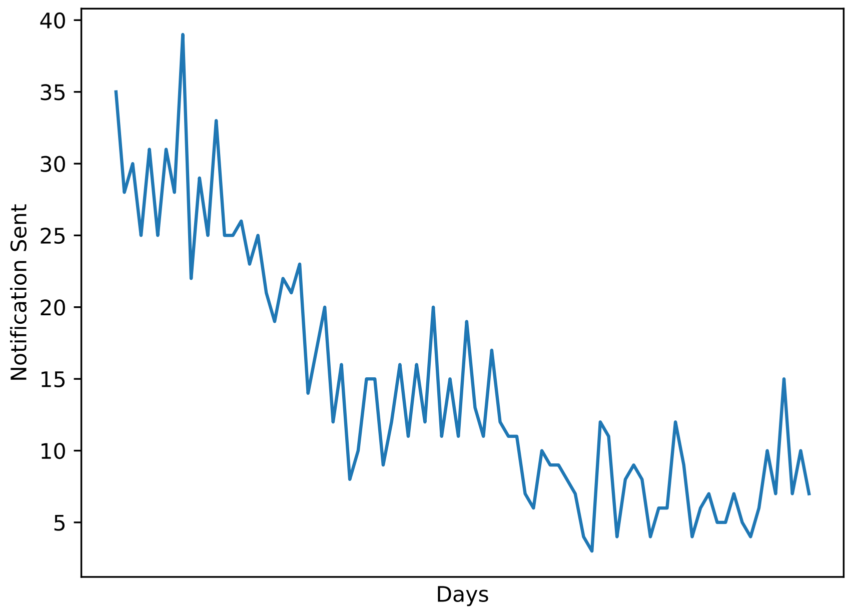


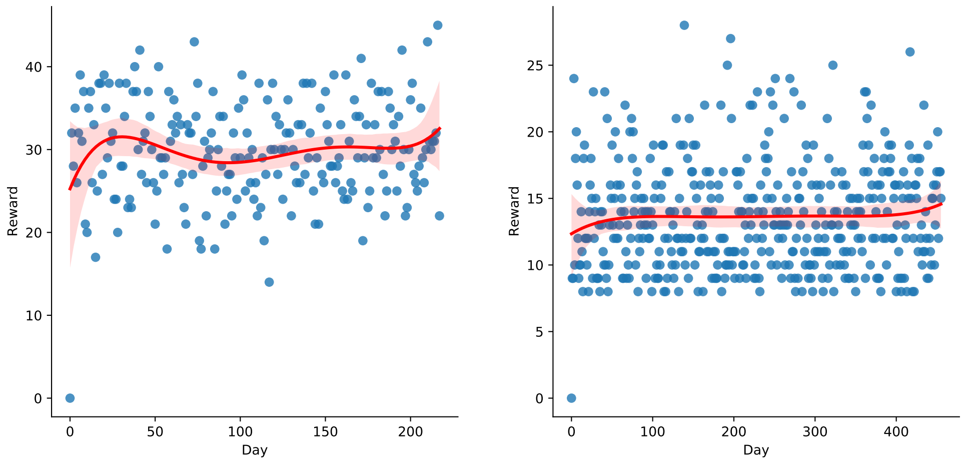
Figure 6.7: Daily notifications sent during training with the DQN agent.

6.3.4 RL vs Random

To prove the Reinforcement Learning agents learn from the environment and increase the cumulative reward, we will compare this with a random agent that does not use the learning but chooses the action randomly. The test is done within the Battery Manager use case and is based on the daily cumulative reward (a day in simulator time steps) and about how many timesteps keep the battery alive.

The evaluation is done with a standard network, whose efficiency has been proved by earlier tests. The use case chosen is the Battery Manager. The network used is the same as the analysis of the Battery Manager use case described before, used in *inference* mode. The random agent act using random action, without doing the training phase.

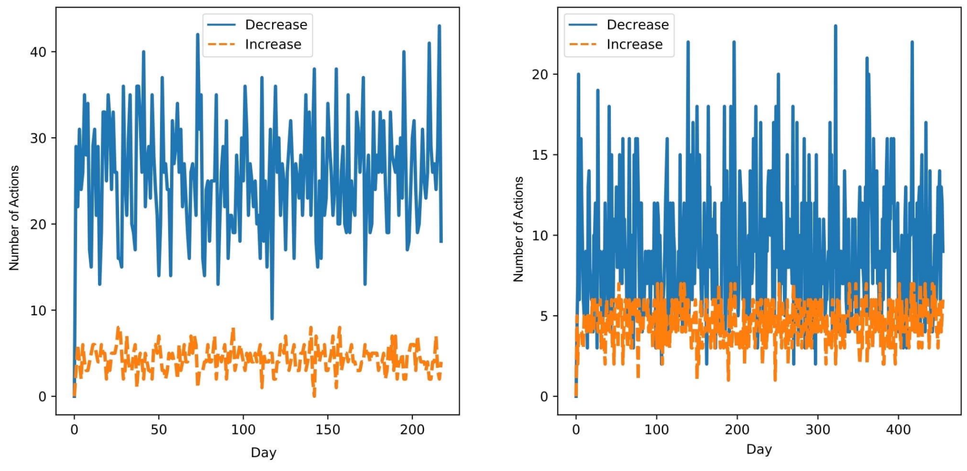
The first plots 6.9a show the daily reward gained by the two agents, the difference is huge in terms of reward. The agent learns how to interact with the environment through the training.



(a) DQN Agent (b) Random Agent

Figure 6.8: Comparison between rewards gained by two agents in the same environment.

The Random Agent stays at the minimum daily reward as shown in the plot 6.9b, while the DQN has a reward 300% better than it. The differences are more evident in the action plot, in which the DQN Agent has a strategy to apply, while the Random Agent does not.



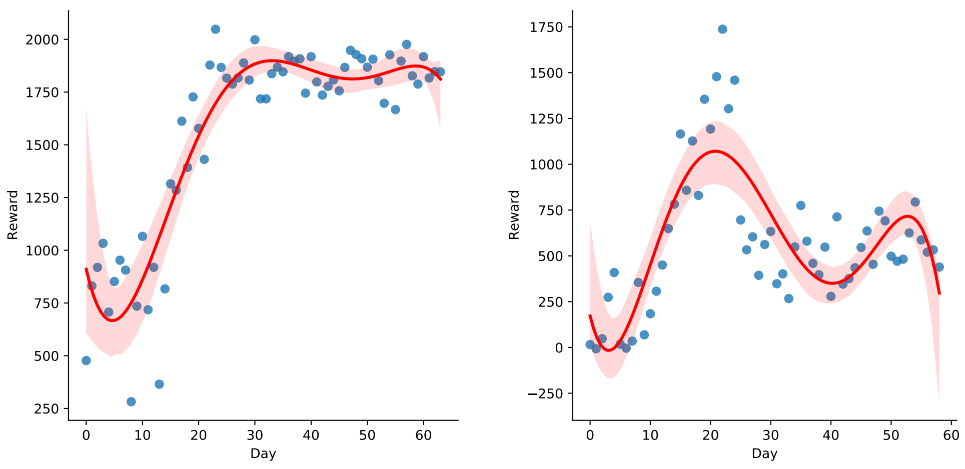
(a) DQN Agent (b) Random Agent

Figure 6.9: Comparison between actions chosen by two agents in the same environment.

The DQN Agent knows, and that proves the framework is fully functioning.

6.3.5 Q-Learning vs DQN

The comparison between the two agents is important to evaluate the potentiality of each algorithm working on-device. The two agents are different from each other and their capabilities are distant, but it is interesting to compare the performance to show the differences in the learning process. To do this test we used Environment 1D. This environment is very easy but it is perfect to show the differences. The Q-Learning moreover is not made to elaborate complex environments and a test with a simple problem is good. The two agents use the same reward system, the Q-Learning has a learning rate of *0.00001* and a gamma of *0.999*, the same as the DQN.



(a) DQN Agent (b) Q-Learning Agent

Figure 6.10: Comparison between two agents in the same environment.

The plot 6.10 shows a very different situation between the two agents. The first graph shows the data of the DQN which reaches and maintains the maximum reward possible in a single episode: 2000. It has a good trend of learning, and after a few days, it can exploit the game. The Q-Learning has a different performance, it reaches its highest nearly at the same time as the DQN, but it is slightly more than half the points. Then the Q-Learning decreases its performance.

6.3.6 DQN Hyperparameter Search

To evaluate the differences between the different network settings we chose to check the performance of the same network, using the hyperparameter to change its behavior. To correctly compare the results the network contains 4 fully-connected layers with different sizes, but at most with 128 units. The use case used is the Notification Manager. The comparison is made with different hyperparameters, keeping as a baseline the hyperparameter used to show the performance in the previous section. These hyperparameters are similar to those seen in the RL literature all the possible parameters seen in the background chapter (number 2.2.1) have an influence on the agent's behavior and they must be carefully tuned to have an optimal learning curve. The use of Reinforcement Learning with neural networks implies using different hyper-parameters w.r.t. to those used in Machine Learning. Starting from the Epochs we chose to fix it at 10 for all the trials, the motivation is the small network the agent will use and because it is a quite common number for this parameter. The learning rates are in line with those used in Machine Learning, the difference between the larger and the smallest ones is the different learning curve. The Gamma is used to fix how the agent must consider the future reward, it is fixed to a high value but it is very important, and also a small change can modify a lot the agent behavior. The batch size is important for the update process as it defines how much data the agent sees during a single update. We resume all the changes in the table 6.6 to be easily visualized, highlighting the differences w.r.t. the baseline for each agent. It is useful to fine-tune the training process to faster the learning and improve the generalization.

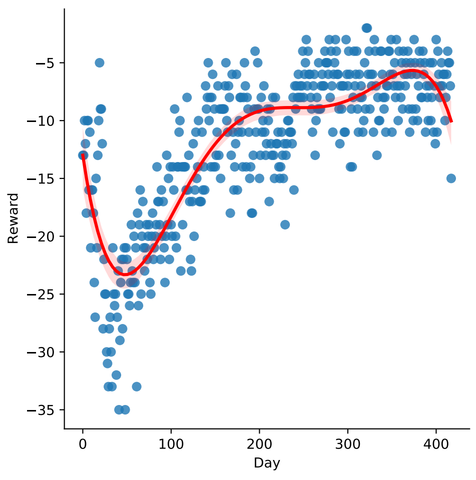
AgentID	**Epochs**	**Learning Rate**	**Gamma**	**Batch Size**
Baseline	10	0.0001	0.999	64
1	10	0.0001	**0.9**	64
2	10	**0.001**	0.999	64
3	10	**0.00001**	0.999	64
4	10	0.0001	0.999	**128**
5	10	0.0001	0.999	**16**

Table 6.6: The different hyperparameters used.

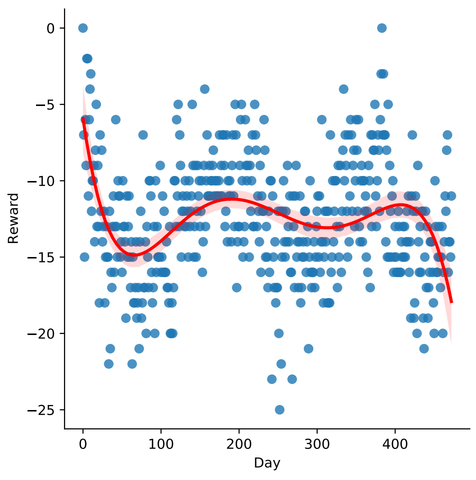
The agent with ID 1 has a different Gamma, this has an impact on the intention of the agent that gives less importance to the future reward than the immediate reward. Agents number 2 and 3 have different learning rates, the former has a bigger learning rate and the latter has a smaller one. This shows the differences between the training process with different learning rates. Agents number 4 and 5 have different batch sizes. They are used to evaluate the best dimension for the batch during training.

The Epsilon Greedy policy is used for all the trials. The epsilon decreases over the time step with a fixed rule: the first 5000 step uses a 0.7 of epsilon, for the next 5000 decreases the epsilon to 0.5 and then uses 0.2 as value. The Exploration during the time decreases to leave the space to the agent's knowledge.

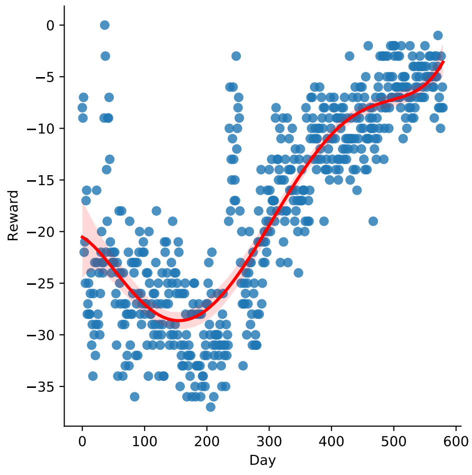
The results are similar to those shown in the Notification Manager section 6.3.3, this is because hyperparameters change the way the network learns and not what it learns. The figure 6.13 below shows the different ways the agent learns with the different hyperparameters. The graphs are created using the cumulative reward for each day. We have not added the labels for the X-axis because the days will not be visible. The average number of days the agents went through is higher than 650.



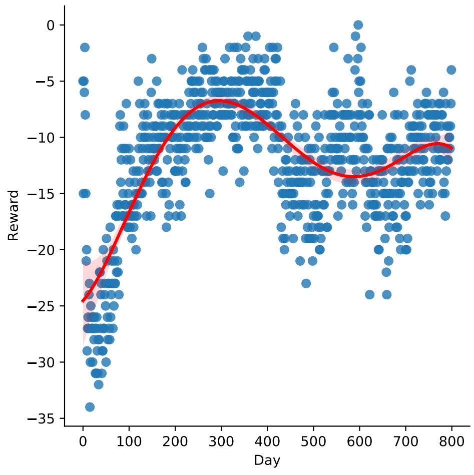
(a) Baseline



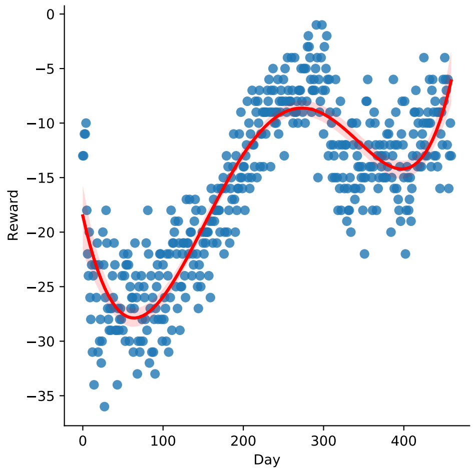
(b) AgentID 1



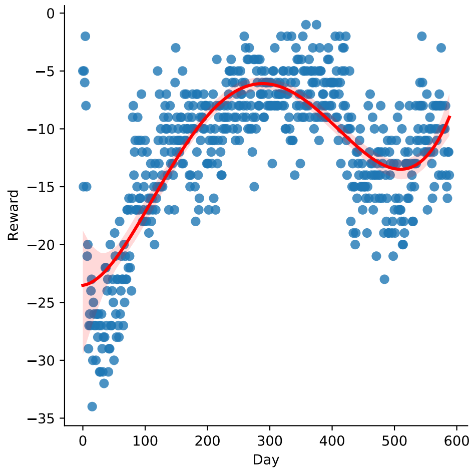
(a) AgentID 2



(b) AgentID 3



(a) AgentID 4



(b) AgentID 5

Figure 6.13: The cumulative daily reward of each agent.

I want to point attention to the first days of each graph, they show a good starting reward due to a fortunate configuration of the network. All the agents start with the same network, that has an initialization of its weights accidentally good for this problem. This is not a problem for the algorithm. All the agents overcame this problem with exploration that decreases the reward before learning. It is a good test for the framework because it shows that it is solid regarding the algorithms. Let's analyze the results, the differences between the agents are visible. The baseline is the one with the best performance if we analyze the total data (fig. 6.11a). The others can have faster learning, for example, the *Agent 3* and the *Agent 5* learn faster and reach the cumulative reward of 0 in fewer epochs, but then degrade and remain at lower rewards (fig. 6.13a and fig. 6.13b). The motivations are different though, the former uses a smaller learning rate w.r.t. the baseline that seems better at the early stages but not optimal to keep the reward high. The latter instead use smaller a batch size (w.r.t. the baseline) which seems a good choice for most of the time but not when the agent runs with smaller epsilon. The *Agent 2* learns more slowly but it has a very interesting increase of the cumulative reward, which stays stable during the following days (fig. 6.12a). The *Agent 4* is very good at the initial time, it is the faster learner to reach a good cumulative reward but at a certain point the performance decrease a lot and it seems unable to return to the same level of performance (fig. 6.13a). The *Agent 2* is the worst one, because the parameter gamma fixed to a smaller value modifies the agent research of the maximum cumulative reward, letting the agent focus on more short-term rewards (fig. 6.12a).

Below is a comparison of the agents on a single graph 6.14. Is is possible to see the big differences on how each agent learns. It is interesting to note the final rewards are all in the similar range.

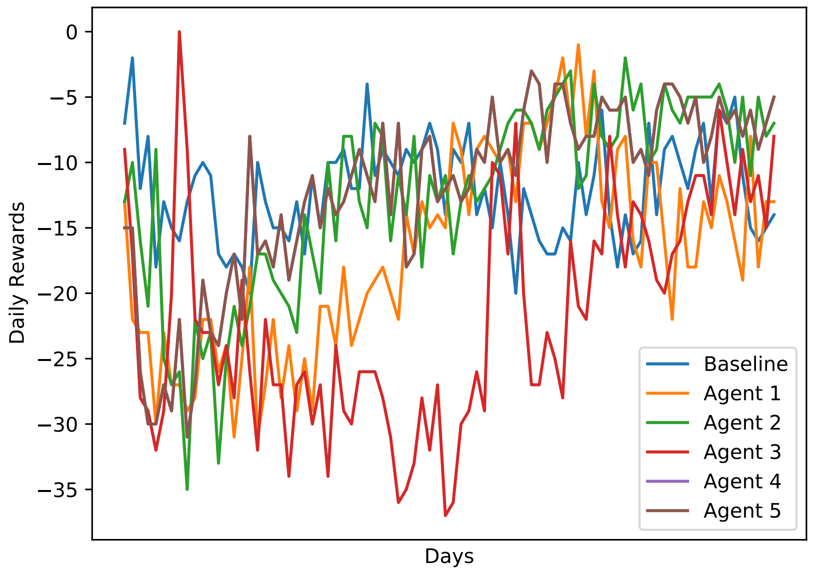
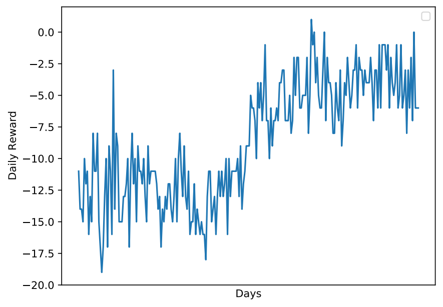


Figure 6.14: Comparison between all the Agents.

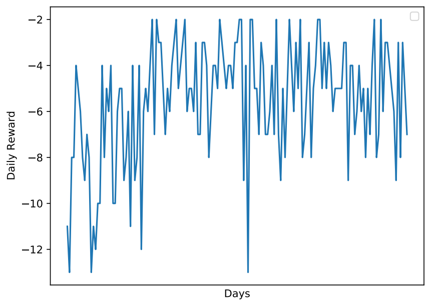
6.3.7 DQN with different sizes

An interesting test is to analyze the differences of different size networks on the same problem. This can help us to check if the device can train different network and made it properly working. To do this test we used the Notification Manager use case and train two agents, one with a smaller network than the one used since here (for the use case) and the other with a bigger network. The hyperparameters are nearly the same for both, the only difference is the batch size fixed to 128 for the bigger network, instead of the default 64. This choice was made to help the bigger network with more normalization. The learning rate is set to 0.00001 and gamma to 0.999, the standard values.

The figure 6.15 shows the differences of learning, the smaller network 6.15a understand the correct behavior and reaches good performance, while the agent with the bigger network 6.15b improves a bit but then stay at sub-optimal reward value. The motivations of this different performance between the two, other than some hyperparameters fine-tuning, could be that the bigger



(a) Agent with small network.



(b) Agent with big network.

Figure 6.15: Comparison between the two networks.

network is oversized for the problem, the problem's complexity is too small for that time of network. In fact, the smaller network reaches good performance also if compared with the standard network used until now.

Chapter 7

Conclusion

The study aims to develop a framework for Reinforcement Learning on iOS devices. The framework developed has some important properties a mobile software must have: stability, easy-to-use, efficiency. We proved these characteristics stand and the framework is fully functioning. But we go on, developing a package that contains a customizable environment, with which the developer can represent its problem with its own approach. The use of custom Sensors, Actions, and Rewards increase the potentiality of this framework exponentially, leaving the developer the freedom to implement its idea. The Environment class becomes the central concept w.r.t. customization, it allows to build an infinite number of different environments the agent can use. The possibility of simply modifying one line of code to change what the observed data are is the key to creating an easy-to-use project. Providing two fully implemented Agents, the package can be used from the basic Reinforcement Learning problems to complex ones, with the switch of the agent. The amount of code necessary to use this package is less than a tenth of the line of code necessary to create all the logic I've implemented. The provided project represents a step towards the use of Reinforcement Learning in daily use applications, an improvement on what the developers can use to increase the personalization in their apps. The Use Cases are providing useful fully working applications, tested with different agents and with good performance, showing the framework has all it needs to be used in real applications, providing useful information about the user's behavior, or changing the way the device manages itself. These examples are used to describe the potentiality of this project. The Battery Manager use case shows how an agent trained on 2 years of simulator time, manages the battery consumption and tries to help the user to do not consume the energy when it is not necessary but without creating inconvenience in the daily use. This use case uses only data retrieved by the device, showing the possibility to manage a device with the right actions and rewards only using internal data. The Notification Manager provides an example of new sensors created and integrated into the environment, while the agent studies the user behavior to exploit the best possible way to send notifications. It is important to show how a new sensor can be detached from the device but it can elaborate the agent and user's data to produce a value the agent can use to learn. This possibility allows the developers to imagine applications that use external sensors to command external objects, with the device only used to elaborate the data (without changing its parameters). The compatibility with all the Apple devices opens the possibility to develop Reinforcement Learning problems on mobile or desktop devices without having big and expensive hardware. The last use case presented is a simple but important environment 1D, to whom the developers can refer to understand the logic of this project. It is not useful in regards to real-world applications but it is very important to give an example that is implemented also in other similar frameworks.

The number of sensors already developed and exposed by this project provides the developers numerous ways to start experimenting, without the need of integrating new sensors. The Sensors provide information about the device, like the battery value, the volume, the screen brightness, but also data about the user, for example, the location or the speed to which the user and the device are moving. The concept of a Sensor can be extended to all the information that can be elaborated by the agent.

In regards to the related works discussed in the relative package, the building of this project can help the researcher to try their ideas with less time spent on programming and more on the test phase, increasing the number of theories and applications a single person can try.

The building of such software could be the starting point of new exciting projects for the devices supported. It opens new ways of customizing the app behavior, it allows to provide personalization to the user in a manner nothing else can. This could become a threat for the user and we will deal with the ethical and privacy implications of this problem.

7.1 Privacy Implications

The goal of this project is to create a framework in which the future developer can experiment with new ways of training its Neural Network. In particular, for this project, the learning technique of Reinforcement Learning is used and developed and this leads to a lot of implications on privacy and ethics. Before deep-diving into these fields, we want to show another important point that is included in this discussion, the possibility to train directly in-phone. Apple allowing to use an updatable model in apps and allowing to train parts of the neural network would open a lot of new possibilities to personalize the behavior of apps ad-hoc for each user.

One of the most important points of this project is the possibility to train the neural network directly on-phone, but differently from what Apple sponsored as APIs to fine-tune the network I've used it to completely update the network in-phone and it's a dramatic change on how personalization can be thought. The fact that all the neural network layers are updated means that the network hasn't any previous knowledge, it is downloaded by the user as a new, with totally random weights, neural network. Applying the Reinforcement Learning technique means having a reward for each action given by the

Agent, this reward can be arbitrarily defined by the developer or, in a more interesting way, it can be based on the users' choices based on their preferences. This latter method brings personalization to a whole new level because the user directly sets the correct behavior and the target of the neural network. In terms of personalization, this is the apex of its application, the final user sets the target for the updates and every final user can build the Agent that has learned and knows the best behavior for itself. Generally speaking, each user that will use an app based on this Package could have, for sure, a different version of the Artificial Intelligence underlying the app. This type of personalization could be seen no more like a single Artificial Intelligence made by society and given to the user but more as a whiteboard with which each final user trains its Artificial Intelligence. There would no longer be a company that pre-trains the network using a lot of data that can be biased or simply not fit for those who use it, each person would build the base knowledge on which the training is done with their mind and behavior. More pragmatically this freedom left to the users would produce a lot of new experiences for him but also for the developer that can imagine an app or a game that changes according to the user's preferences, but on the other hand, could lead to Artificial Intelligence that can learn bad behaviors. We know that the usual training dataset is composed of a huge amount of data and is still biased or erroneous in certain situations. In a case in which the network is personalized, and so trained, by a single human (or more than one but for sure less than the number of people that built the massive dataset we usually consider) the bias would become important w.r.t. a supposed normal distribution of data. We can also say that the Neural Network could become non-generic in a broad sense, meaning after a lot of time the network would know, with a certain amount of confidence, the usual behavior of a single user (or more users, but limited as before) and it can't be used with the same accuracy by others users mostly because of differences between every user behavior and preferences. The discussion over the Ethics of this learning method is becoming too big to be dealt with here.

This project aims to allow developers to train neural networks through Reinforcement Learning directly on iPhone.

What we described before in terms of Ethics has a strict correlation to the argument of Privacy. The fact that each user could train its neural network directly, with its preferences from scratch, would make a huge difference in how the training is done compared to normal training. Also in Reinforcement Learning usually the environment with which the agent is trained is a simulation over static data, it can be trained using a car simulator or observation of sensors or analytical data but in none of them, the final user has any sort of ability to edit the data based on its preferences. Usually, in Machine Learning, the personalization part of the training means starting with a fully trained network and then updating only a small part of it with newly collected data. To deep dive into this situation, we can think that most of the knowledge is already built-in in the network before the user becomes part of the training; this implies much less user engagement and so less personalization. In terms of privacy how and where the training is done are becoming issues because training a network implies using data, and data need to be collected somewhere. If the data are collected directly on the phone the data are strongly correlated to the final user and this is a privacy issue. We know that our smartphones collect data continuously about us to perform some kind of analytics and suggest, for example, the best restaurant near us based on what we like to eat but there are lots of rules that regulate that data, fortunately. These rules impose a lot of limits on processing the data collected especially because to elaborate those data companies need to send them to their servers all around the globe, and where stays under different national rules where the servers reside that could be good or bad. This project trains the network directly on phone, so there is no need to send the data around. This point is a huge change in the neural network training paradigm applied to smartphones. If we can collect the data in the same place in which we train the network we have the certainty that no one else can see the data, so nobody can learn from it apart from the neural network. At this point, the data are used directly to train and then can be deleted because they become useless. The knowledge extracted from the neural network is fully correlated to the user behavior and this is an issue for privacy because if someone stole the weights of the net from the user he would have a lot of information about the user. Within this project, the network is initially downloaded from the user with totally random weights and then stored inside the phone. If the developer can assure a sort of security level to gain the access to the network, we can have the certainty that not only the data collected but also the knowledge of the network, that if we may say so is more connected to the user, to stay on phone and only there. If this situation could be created then we would have the maximum level of privacy in which no one but the user could have access to the knowledge, and the data would be used only by the network.

7.2 Limitations

Developing this package we have encountered a lot of problems, basically, we have worked without having the proper tools to do that. Apple is a brand that makes devices and exposes APIs to build applications for the users such as games or productivity apps or utility apps.

7.2.1 Code Limits

It has shown recently the new CoreML kit used to deploy the model to do inference on-device, and even more recently it has produced the Neural Engine, a processor made only for Machine Learning tasks. Apple provides a lot of features and services for those who develop applications but, rightly, it doesn't develop anything to ease the work of studying something new. Everyone also knows that Apple is strict in giving the developer permission about its products, especially if compared with Google and its mobile O.S. Android. The limits of my package are mostly delimited by the policy set by Apple about what an application can do. For example, an application that manages the battery, changing the settings of the phone may be a good application especially if it is customized on every single user. Unfortunately, as already said, it is not possible to modify the settings without the user intervention, if not when the app is in the foreground. Android leaves the possibility to modify every single thing as a developer, losing a bit of security that is the point of Apple. Another limit is correlated to the previous one and is intrinsic in the programming language used: Swift. Swift is a relatively new programming language that has the purpose to work in symbiosis as much as possible with Apple devices. It has characteristics similar to whose of Objective-C on which it is inspired, but it has fewer features than its father. Therefore, usually in Machine Learning Python is used to develop and run the models but Python is not a language that can run on a device easily, firstly because it is interpreted and not compiled, as opposed to Swift.

We have already dealt with the limits of the Foreground and Background processes (chapter 5.2) but they need a mention here. The Background process can be implemented by using two different classes, the App Refresh Task and the Processing Task. The former has the limit of 30 seconds and it was created with the purpose of refreshing the content of the app; the latter does not have any limit and can be used to process data, as well as download external data like a network.

The public API exposed by Apple for Machine Learning includes a lot of features and models that are already built to be used directly on the phone. This kit, named CoreML, uses a new type of file called mlmodel that describes in an Apple way the architecture and works of a model. These types of files can be downloaded from the Apple site or can be created by developers using Python and Turi Create and then included as they are into the iOS App. This approach limits the development of custom models, in fact, adds the necessity of using a Python script to create a mlmodel that then can be trained on-device.

Actually, to the best of my knowledge, there is no API to create a mlmodel directly with Swift, it's necessary to use Python and Turi Create. A solution can develop a Python script that the developer needs to call to create the desired model, the official API describes that in Swift is possible to directly import the built model by Apple, though they are very application-specific, e.g. models for image classification, sound reconnaissance and so on. The script made includes initially 2 Reinforcement Learning algorithms: Q-Learning and Deep Q-Network, all three customizable defining the parameter.

7.2.2 Device Limits

The biggest part of the limits belongs to the programming language and the APIs not created for the purpose of training a whole network on-device or for Reinforcement Learning. Other limits belong to the devices, if in the normal application of RL using a computer the limit is the memory and the computational power, then also in a mobile device environment the same problems arise. A limit on the network size is fixed by Apple, it is impossible (or blocked) to train a network larger than 30 layers. The number of parameters on the contrary seems to have no limits, if not the computational power of the device. In case of a high number of parameters, the device will update the network using a lot of time instead of breaking apart. Without limits on the training process, it is a developer's choice how to implement the networks, but they have to take care of the device and the usability of the device during the training. If they use a big network with a lot of parameters, but that stays within the limits, then when the network update begins the phone will slow down and warm up a lot. The developer has to be careful to do not to ruin the users' device.

A problem directly connected with the previous one is the use of PythonKit, a library made by Apple that allows the developer to use Python in a swift file. The use of it would make it easier to create ML model thanks to the use of Python library made for this scope. Unfortunately on iOS and WatchOS, and more generally on iOS app isn't permitted to use so it's limited to macOS. Another approach allowed by PythonKit is to call Python script from a Swift script, implicitly this approach include the use of Python script into the Xcode project, so a solution for creating a custom model on the fly could be to call a Python script to create the model but a limit from Apple is no Python file in iOS app.

7.3 Future Work

The package presented in this study has been created from scratch, with almost no other projects to compare, so the approach may or not be correct. All the concepts included are developed only by me, but usually, a project like this involves a team with different professional figures that can help design the package. If we have to think about what we was not able to do, for lack of knowledge or lack of time, we would say the limits imposed by the programming language and by Apple are the first things we will try to overcome. But many other features can be included in this project, and they can be improved. Below are some ideas. An idea for a new feature could be to allow the developer the Federated Learning: a technique that enables mobile phones to collaboratively learn a shared prediction model while keeping all the training data on the device, decoupling the ability to do Machine Learning from the need to store the data in the cloud. With a better knowledge of Swift maybe it is possible to expand and make parameterizable all the classes I've built. This is one of the most important future works we can think of, because it leaves to the developer the possibility of using more data types instead of only Double, and this could be a huge game-changer. The use of images is allowed with the conversion of the image into a vector of doubles, but the Apple API allows to use of the images directly as input of the network. This is not supported yet, because it would need a different architecture as well as a different update process.

In this framework are included two agents, one is Q-Learning and the other is Deep Q-Network, but in literature, there are more. The development of the other algorithms as Agents is one of the most important features to allow developers to choose the best agent to fit their problems. For example, the implementation of a Monte Carlo Method or a more complex and recently proposed one, e.g. the Proximal Policy Optimization [18]. The use of files to store the data is the simplest method, but many more are better in terms of space or management. The best idea could be to use the SQL Database provided by Apple to manage the data. This will have a lower impact on the device memory usage during the training phase, and also a great impact while storing the data in the observation phase.

Appendix A

Technical Appendix

Swift RL is a new package to help the developer implement Reinforcement Learning on mobile devices. The package, with all the examples and the documentation, is published on GitHub at the URL: `https://github.com/pavva94/SwiftRLLib`. This document will provide an extensive description of the APIs developed. Swift RL is based on the literature, the main concepts are the **Environment** (see A.1) and the **Agent** (see A.2). Both can be configured and personalized by the developer. The package includes different classes, protocols, and functions to be used by the developers to develop their ideas. In this appendix we will extensively describe the APIs developed, the developer must find complete documentation that helps in the first approach. At the package URL, it is published complete documentation in DocC format, the official one for the Apple documentation.

The Swift package can be easily imported into any project with the appropriate function of Xcode.

A.1 Environment

A.1.1 Initializer

The Environment is a customizable object, in the instantiation it needs:

```
public init(
    observableData: [String],
    actions: [Action],
    rewards: [Reward])
```

The list of strings called **observableData** (see A.3.4) will contain the names of each sensor selected from the list of sensors provided. The **actions** (see A.3.1) need a list of actions, the object that contains the real action to execute. The **rewards** (see A.3.2) is the list of rewards implemented by the developer.

A.1.2 Methods

act: the function execute the exec() function of the Action choose.

```
func act(state: RLStateType, action: RLActionType)
```

addObservableData: the function add an ObservableData to the list, in the last position. This is used to attach new sensors to the environment.

```
func addObservableData(s: ObservableData)
```

read: the function is called by the environment to call the read() function for each ObservableData given. The returned data is a list of Double.

```
func read(fromAction: Bool) -> RLStateType
```

reward: the function returns the state size of the Environment calculated by adding the single state sizes of each sensor.

```
func reward(
    state: RLStateType,
    action: RLActionType,
    nextState: RLStateType
) -> Double
```

getActionSize: the function returns the action size of the Environment, set by user. The returned data is an integer.

```
func getActionSize() -> Int
```

getStateSize: the function returns the state size of the Environment calculated by adding the single state sizes of each sensor. The returned data is an integer.

```
func getStateSize() -> Int
```

A.2 Agent

A.2.1 Initializers

The Agent is a customizable object, in the instantiation it needs:

```
init(env: Env,
policy: Policy,
parameters: Dictionary<ModelParameters, Any>)
```

The **env** is the environment previously created (see A.1), the **policy** is the policy object previously created (see A.3.3). The parameters are optional and can contain different keys to modify the agent's behavior.

A.2.2 ModelParameter

The parameters used to modify the agent's behavior for the observe and training processes.

- **agentID**: identifier of the agent.
- **batchSize**: batch size for the training, from the list [8, 16, 32, 64, 128, 256].
- **bufferPath**: path of the Buffer.
- **databasePath**: path of the database.
- **episodeEnd**: function to define the end of the episode.
- **epochs**: number of epochs used in training, from the list [1, 10, 50].
- **gamma**: gamma value for the update process.
- **learning_rate**: learning rate value.
- **secondsObserveProcess**: seconds between two call of the Observe process.
- **secondsTrainProcess**: seconds between two call of the Train process.
- **trainingSetSize**: training set size.

A.2.3 Methods

start: the function start the agent's processes based on the given parameters.

```
func start(WorkMode, AgentMode)
```

A.2.4 Parameters

WorkMode: the Work Mode defines how the agent will works.

- *background*: the agent uses the background processes to interact and learn.
- *timer*: the agent uses the timers to interact and learn.
- *both*: the agent uses both.

AgentMode: describes the Agent Mode

- *inference*: the agent works in inference mode, without training.
- *training*: the agent trains itself while interacting.

The Agent has two subclasses: **Q-Learning** and **DeepQNetwork**. Both have inside the implementation of the respective algorithms and include the management of the file, the knowledge upload, and the action selection.

A.3 Objects

In this section, we will show the protocols and the classes used to provide a standard structure for each necessary object.

A.3.1 Action

The Action is a protocol that defines the structure of the object in which the developer inserts the action the agent will do. The developer must inherits from this protocol to define the action object to be passed to the environment. The protocol has some required properties and a function to be implemented.

```
public protocol Action {
    var id: Int { get }
    var description: String { get }
    func exec()
}
```

Example actions are provided but useless in terms of usability due to the strict correlation to the problem the developer is approaching.

A.3.2 Reward

The Reward is a protocol that defines the structure of the object in which the developer inserts the reward the agent will receive for its actions. The developer must inherit from this protocol to define the reward object to be passed to the environment. The protocol has some required properties and a function to be implemented.

```
public protocol Reward {
    var id: Int { get }
    var description: String { get }

    func exec(
        state: RLStateType,
        action: RLActionType,
        nextState: RLRewardType
    ) -> Double
}
```

Example actions are provided but useless in terms of usability due to the strict correlation to the problem the developer is approaching.

A.3.3 Policy

The Reward is a protocol that defines the structure of the object in which the developer inserts the reward the agent will receive for its actions. The developer must inherits from this protocol to define the policy object to be passed to the agent that will defined how they will act. The protocol has some required properties and a function to be implemented.

```
public protocol Policy {
    var id: Int { get }
    var description: String { get }
    func exec(model: MLModel, state: MLMultiArray) -> Int
}
```

The package provides a **Random Policy** and an **Epsilon Greedy Policy** that can be used.

A.3.4 ObservableData

The RewarObservableData is a class that defines the structure of the object the developers will inherit for their new sensors. The protocol has some required properties and a function to be implemented.

```
open class ObservableData {
    var name: String
    var stateSize: Int

    public init(
        name : String = "ObservableData",
        stateSize: Int)

    open func read(
        _ state: RLStateType = []
    ) -> RLStateType
}
```

The package provide different ObservableData:

- **AltitudeSensor**: the altitude sensor returns a value indicating the altitude is in the unit of measure of the device.
- **BarometerSensor**: the barometer sensor returns the pressure value in the unit of measure of the device.
- **BatterySensor**: the battery sensor returns values from 0.0 to 100.0, based on the battery value.

- **BrightnessSensor**: the brightness sensor returns values from 0.0 to 1.0, based on the screen brightness of the device.
- **ClockSensor**: the clock sensor returns the hours, minutes, and seconds.
- **DateSensor**: the date sensor returns the current day, the current month, and the current year.
- **GyroscopeSensor**: the Gyroscope sensor returns the X-axis, Y-Axis, and Z-axis, using the gyroscope hardware.
- **HourSensor**: the hour sensor returns the current hour.
- **LockedSensor**: the altitude sensor returns 1 or 0 if the device is locked or not.
- **LowPowerModeSensor**: the altitude sensor returns 1 or 0 if the low power mode is active or not.
- **MinuteSensor**: the altitude sensor returns the current minutes.
- **OrientationSensor**: the orientation sensor returns 0 or 1 based on the orientation of the device: Landscape or Portrait.
- **SecondSensor**: the second sensor returns the current seconds.
- **SpeedSensor**: the speed sensor returns a value for the current speed in the unit of measure of the device.
- **VolumeSensor**: the volume sensor returns a value indicating the volume of the device.

A.4 Datatypes

Different datatypes are used in this project the developer should know.

- **Sarsa** Tuple: is the tuple containing the state S_t, the action A_t, the reward R_t and the next state S_{t+1}.
- **RLStateType**: the datatype for the state, typealias of [Double].
- **RLActionType**: the datatype for the state, typealias of Int.
- **RLRewardType**: the datatype for the state, typealias of Double.

The Sarsa Tuple that is the tuple containing the state S_t, the action A_t, the reward R_t and the next state S_{t+1}. But also the type alias used for the state, action, and reward type: RLStateType, RLActionType, and RLRewardType. The use of type alias helps the cleanness of the code.

Bibliography

[1] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce. Reinforcement learning for android gui testing. *Conference: the 9th ACM SIGSOFT International Workshop*, 2018. URL: `https://doi.org/10.1145/3278186.3278187`.

[2] O. Arafat and D. Riehle. The comment density of open source software code. *2009 31st International Conference on Software Engineering - Companion Volume, ICSE 2009*:195–198, May 2009. DOI: `10.1109/ICSE-COMPANION.2009.5070980`.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. arXiv: `1606.01540 [cs.LG]`.

[4] L. Cai, M. Boukhechba, N. Kaur, C. Wu, L. E. Barnes, and M. S. Gerber. Adaptive passive mobile sensing using reinforcement learning. In *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–6, 2019. DOI: `10.1109/WoWMoM.2019.8792967`.

[5] M. Campolese. Swift code metrics. 2022. URL: `https://github.com/matsoftware/swift-code-metrics` (visited on 02/21/2022).

[6] A. Choudhary. A hands-on introduction to deep q-learning using openai gym in python. 2019. URL: `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/` (visited on 03/04/2022).

[7] X. Dai, I. Spasić, B. Meyer, S. Chapman, and F. Andres. Machine learning on mobile: an on-device inference app for skin cancer detection. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 301–305, June 2019. DOI: `10.1109/FMEC.2019.8795362`.

[8] Y. Deng. Deep learning on mobile devices: a review. In S. S. Agaian, V. K. Asari, and S. P. DelMarco, editors, *Mobile Multimedia/Image Processing, Security, and Applications 2019*, volume 10993, pages 52–66. International Society for Optics and Photonics, SPIE, 2019. DOI: `10.1117/12.2518469`. URL: `https://doi.org/10.1117/12.2518469`.

[9] S. Flutura, A. Seiderer, I. Aslan, C.-T. Dang, R. Schwarz, D. Schiller, and E. André. Drinkwatch: a mobile wellbeing application based on interactive and cooperative machine learning. In *Proceedings of the 2018 International Conference on Digital Health*, DH '18, pages 65–74, Lyon, France. Association for Computing Machinery, 2018. ISBN: 9781450364935. DOI: `10.1145/3194658.3194666`. URL: `https://doi.org/10.1145/3194658.3194666`.

[10] J. Han and S. Lee. Performance improvement of linux cpu scheduler using policy gradient reinforcement learning for android smartphones. *IEEE Access*, 8:11031–11045, 2020. DOI: `10.1109/ACCESS.2020.2965548`.

[11] N. Islam, S. Das, and Y. Chen. On-device mobile phone security exploits machine learning. *IEEE Pervasive Computing*, 16(2):92–96, 2017. DOI: `10.1109/MPRV.2017.26`.

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019. arXiv: `1509.02971 [cs.LG]`.

[13] A. LLC. Apple Core ML. 2022. URL: `https://developer.apple.com/documentation/coreml` (visited on 02/20/2022).

[14] G. LLC. Google ML Kit. 2022. URL: `https://developers.google.com/ml-kit` (visited on 02/20/2022).

[15] R. C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017. ISBN: 978-0-13-449416-6.

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: `https://www.tensorflow.org/`. Software available from tensorflow.org.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop 2013*, 2013. arXiv: `1312.5602 [cs.LG]`.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL: `http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17`.

[19] Z. Shen, K. Yang, Z. Xi, J. Zou, and W. Du. Deepapp: a deep reinforcement learning framework for mobile application usage prediction.

IEEE Transactions on Mobile Computing:1–1, 2021. DOI: 10.1109/TMC.2021.3093619.

[20] M. Shvo, Z. Hu, R. T. Icarte, I. Mohomed, A. Jepson, and S. A. McIlraith. Appbuddy: learning to accomplish tasks in mobile apps via reinforcement learning, 2021. arXiv: 2106.00133 [cs.AI].

[21] TinyML. Tiny ML. 2022. URL: https://www.tinyml.org (visited on 02/20/2022).

[22] D. Toyama, P. Hamel, A. Gergely, G. Comanici, A. Glaese, Z. Ahmed, T. Jackson, S. Mourad, and D. Precup. Androidenv: a reinforcement learning platform for android, 2021. arXiv: 2105.13231 [cs.LG].

[23] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. P. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017. arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782.

[24] P. Voigt and A. v. d. Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017. ISBN: 3319579584.

[25] D. Weir, S. Rogers, R. Murray-Smith, and M. Löchtefeld. *A user-specific machine learning approach for improving touch accuracy on mobile devices*. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 2012, pages 465–476. ISBN: 9781450315807. URL: https://doi.org/10.1145/2380116.2380175.

Acknowledgements

I would to thanks heartfully Professor Mirco Musolesi for all he has done for me about this project. Your presence was fundamental to building this project and your guidance has untangled a lot of doubt. I hope you liked my work, it was a pleasure to work together. I have to be grateful to the University of Bologna to have created this master's course, which was very interesting even in its first year.

Federico, you were my colleague for most of the time I spent at this university, we developed a lot of projects together, we are the best team. Thanks for all the time spent with me, it was special. I hope one day to work with you, ”like the old days”. Andrea, Michele we spent less time together but I feel always at home. I wish all of you the best.

Ettore, Antonella, Erika, Noemi, Siro, Marisa, Carlo, Danilo, Elisabetta; my goergeus family. You are so important in my life, you are my inspiration. I made this for you. Whatever the future will be I will miss you a lot more than I would like to say. Simone, Alessandro; you listened to me so much time I think you know better than me Artificial Intelligence. You are always ready to listen, always interested, I love to talk with you guys. Even on the worst days, you make me laugh. Many other people I have to thank, all of you push me here somehow; everyone offering help whenever it needs, but more importantly enjoying my life with you is a pleasure.

Thanks to all.

A Baldo.