Tartu University

Faculty of Science and Technology

Institute of Technology

Lilou Gras

Exploring Smartphone-Based Reinforcement Learning Control for Educational Robotics: Implementation on OpenBot

Master's thesis (30 EAP)
Robotics and Computer Engineering

Supervisor: Dr. Naveed Muhammad

Tartu 2023

Resümee/Abstract

Nutitelefonil põhineva tugevdusõppe juhtimise uurimine haridusrobotikas: rakendamine OpenBotil
See uurib võimalust rakendada Tugevdamisõppe (RL) algoritme täielikult nutitelefonis, et juhtida hariduslikku robotplatvormi OpenBot. Selle uuringu eesmärk on välja selgitada, kas RL-i saab teostada Android-nutitelefonides ilma simuleeritud keskkondadeta ja kas see oleks õpilastele ja entusiastidele praktilise RL-projektina kättesaadav. Algselt testiti Sügav Q-Õpe (DQL) ja Poliitikagradiendi (PG) algoritme standardsete RL-stsenaariumide Cartpole ja Pong abil. See võimaldas saada ülevaate mõlemast algoritmist ja sellest, mida edukas RL-koolituses oodata. Seejärel rakendati poliitikagradiendi algoritm täielikult OpenBoti juhtivas nutitelefonis, et sõita 15 sekundi jooksul üle raja. Üldiselt suudab agent pärast ligikaudu 400 poliitikagradienti kasutavat koolitusepisoodi edukalt navigeerida rajal sihitud 15 sekundi jooksul pooltel katsetest. Vaatamata uuringu julgustavatele tulemustele on mõned tehnilised väljakutsed endiselt lahtised, nagu plahvatuslikud gradiendid, kaalu lähtestamise juhuslikkus ja inseneriprobleemid, nagu suur akukulu.[1]

CERCS: T125 Automatiseerimine, robootika, juhtimistehnika; P176 Tehisintellekt

Märksõnad: Tugevdusõpe, kontroll, robootika, openbot, poliitikagradiendiv

Exploring Smartphone-Based Reinforcement Learning Control for Educational Robotics: Implementation on OpenBot
This research explores the feasibility of implementing Reinforcement Learning (RL) algorithms entirely on a smartphone to control an educational robotic platform, OpenBot. This study aims to determine if RL can be executed on Android smartphones without simulated environments and whether it would be accessible for students and enthusiasts as a practical RL project. Initially, Deep Q-Learning (DQL) and Policy Gradient (PG) algorithms were tested on standard RL scenarios, Cartpole and Pong. This allowed to gain insights on both algorithms and what to expect in a successful RL training. The policy gradient algorithm was then implemented entirely on the smartphone controlling OpenBot to drive across a track for 15 seconds. In general, after approximately 400 episodes of training using policy gradient, the agent was able to successfully navigate the track for the aimed 15 seconds in half of its attempts. Despite the encouraging results of the study, some technical challenges remain open, such as, exploding gradients, the randomness of weight initialization, and engineering challenges such as high battery consumption.

CERCS: T125 Automation, robotics, control engineering; P176 Artificial Intelligence

Keywords: reinforcement learning, control, robotics, openbot, policy gradient

[1]The abstract was translated by Anastasia from Fiverr.

Contents

Resümee/Abstract **2**

List of Figures **6**

Abbreviations. Constants. Generic Terms **8**

1 Introduction **9**
1.1 Context . . . 9
1.2 Problem Statement and Objectives . . . 9
1.3 Structure of the Manuscript . . . 10

2 Introduction to Reinforcement Learning **12**
2.1 Origin and Application . . . 12
2.2 First Steps into Reinforcement Learning . . . 13
2.2.1 Introduction to terms used in Reinforcement Learning . . . 13
2.2.2 Markov Decision Processes . . . 15
2.2.3 Bellman equations . . . 16
2.2.4 Exploration vs Exploitation in Reinforcement Learning . . . 17
2.3 Challenges with Reinforcement Learning . . . 18
2.3.1 Challenges of Implementing Reinforcement Learning on Hardware-Constrained Platforms . . . 18
2.3.2 Challenges of Implementing Reinforcement Learning on a Smartphone 19

3 Reinforcement Learning Methods and Algorithms Studied in this Thesis **20**
3.1 Q-Learning . . . 20
3.2 Artificial Neural Networks . . . 21
3.3 Deep Q-Learning . . . 23
3.4 Policy Gradient . . . 24

4 Presentation of OpenBot **25**
4.1 Description of OpenBot . . . 25
4.1.1 Hardware Components . . . 25
4.1.2 Mechanical Assembly & Electrical Configuration . . . 26
4.2 Presentation of the App . . . 26
4.3 Introduction to the Server and Python Scripts . . . 27
4.3.1 Python Server . . . 27
4.3.2 Python Scripts . . . 29

5 Foundational Experiments **30**
5.1 Preliminary Attempt at Implementing Reinforcement Learning on OpenBot . . 30
5.1.1 Experiment Description and Objectives 30
5.1.2 Adding the OpenCV library to Android Studio and OpenBot's Base Code 31
5.1.3 Creating the new Feature 31
5.1.4 Implementation of the Reward: 32
5.1.5 Experiment Protocol 35
5.1.6 Results 36
5.1.7 Conclusion and Discussion 38
5.2 Implementing Reinforcement Learning Algorithms In Different Scenarios . . . 39
5.2.1 Cartpole 39
5.2.2 Pong 42
5.2.3 How These Examples Help Prepare for the Implementation of Reinforcement Learning on OpenBot 48

6 Methodology to Implement Reinforcement Learning on OpenBot **49**
6.1 Updated Experiment and Objectives 49
6.2 Implementing Reinforcement Learning Features 51
6.2.1 State 51
6.2.2 Reward 52
6.2.3 Actions and Termination 52
6.2.4 Epsilon-greedy 53
6.3 Creating the Model and Implementing Policy Gradient 54
6.3.1 Model initialization 54
6.3.2 Model Forward and Backward pass 54
6.3.3 Gradient Ascent 54
6.3.4 Training Process 55
6.4 Difficulties Arising with Implementation 56
6.4.1 Using ArrayLists 56
6.4.2 Errors with the Random Action 57
6.4.3 Crashing if Restarting too Fast 57
6.4.4 Time and Battery Consumption 57
6.4.5 Exploding Gradient 57
6.5 Protocol 58

7 Ensuring Safety of the Robot and its Surrounding **59**
7.0.1 Manual Emergency Stop Using Controller 59
7.0.2 Automatic Stop When Exiting the Open Circuit 59
7.0.3 Pre-existing feature on OpenBot 59

8 Results **60**
8.1 Training Duration and Variability 60
8.2 Learning Progress and Performance Milestones 60
8.3 Crashing and Losing Models 60

9 Discussion **62**
9.1 Interpretation of the Results 62
9.2 Limitations of the Study 63
9.3 Recommendations for Future Research or Work 63

9.4 Additional Notes 63

10 Conclusion 65

Bibliography 67

Appendices 70

10.1 External Tools Used in the Making of this Thesis 70

10.2 Implementing OpenCV on Android Studio 70

10.3 Forward and Backward Policy 72

Non-exclusive license 74

List of Figures

2.1 Illustration of the Cat (agent) situation and its two possible action. 14
2.2 Graphical representation of epsilon evolution over 10000 episodes with initial ϵ set to 0.15 and decay rate to 0.999. 18

3.1 Simplified Artificial Neural Network with a picture of a cat of size 128x256x3 as the input, followed by an undetermined number of hidden layers and nodes and finally the output layer with two nodes; probability of the image to be a dog and probability to be a cat. In between each layer, there are Weights connecting each node to all the following nodes of the next layer. 22

4.1 Picture of OpenBot Ready-To-Run model with smartphone mounted. From OpenBot's GitHub [28]. 25
4.2 Wiring diagram. Electrical connections between battery, motor driver, micro-controller, speed sensors, sonar sensor and indicator LEDs. From the OpenBot GitHub and paper [27] [28]. 26
4.3 Preview of the data collected with pictures flashed consecutively like a video, the option to move directory is also displayed as 'Actions'. 27
4.4 Screenshot of the server's model uploading scheme. The same model can be uploaded multiple times under different names and uploaded models can be deleted. 28
4.5 Screenshot of the models details, the model is uplaoded to the OpenBot app by clicking the 'Push to Phone' button. 28

5.1 Illustration of the reinforcement learning process in the line following scenario on the left and picture of the circuit on the right. 31
5.2 Display of photos before and after processing to detect the line to follow. On the left is displayed the ideal result of line detection. On the right is example on how the luminosity impacts the result and could be problematic. 33
5.3 Simplified Flow Chart of the Reinforcement Learning protocol in the preliminary experiment. 36
5.4 Screenshot of the Cartpole game with a cart attached on an horizontal axis and the pole vertically standing above. 39
5.5 Data flow of the algorithm for training a DQN agent to play cartpole. Diagram taken from the tutorial [34]. 41
5.6 Graphs of 4 different instance of training the model on Cartpole, with the same hyperparameters for each, showing how training varies from one training to another. 42

5.7 Picture of Pong-v0 from the OpenAI gymnasium library [33] rendered in rgb_array. On the left, in brown, is the computer simply following the ball and playing against the agent in green, on the right. 43
5.8 Illustration of the ANN model trained to play Pong. Image from Andrej Karpathy's blog [37]. 45
5.9 Plot showing the moving average of rewards over the last 100 episodes during the 50k-episode training. 46
5.10 Graphical representation of the performance of the two models, in blue the original and in orange the modified model with two outputs. Performance is calculated using the reward moving average of the last 100 episodes and the models were trained on 15k episodes. 48

6.1 Picture of an extremity of the open circuit, displaying the setup of a black plastic bag over a clear floor and white tape to structure the track. 50
6.2 Screenshot of the new policy gradient feature added when first opening the OpenBot app. 51
6.3 Processed image received by the model, the image is of size 128x30. 51
6.4 Screenshot of the UI on a Redmi Note 11 of the POLICYGRADIENT feature showing the different possible interactions such as the possibility to enter an epsilon value and to toggle the resume training. 53
6.5 Simplified Flow Chart of an episode in this experiment. 56

Abbreviations, Constants, Generic Terms

AI - Artificial Intelligence

ANNs - Artificial Neural Networks

app - Mobile Application

CPU - Central Processing Unit

DQL - Deep Q-Learning

DRL - Deep Reinforcement Learning

GPU - Graphics Processing Unit

ML - Machine Learning

MDP - Markov Decision Process

MoR - Mean of Rewards

MSE - Mean Squared Error

PG - Policy Gradient

PS4 - PlayStation 4

OTG - On-The-Go

RL - Reinforcement Learning

RMSProp - Root Mean Square Propagation

ROS - Robotic Operating System

RTR - Ready-To-Run

UI - User Interface

1 Introduction

1.1 Context

With the current rapid evolution of Artificial Intelligence and Machine Learning (ML), Reinforcement Learning (RL) is becoming increasingly sophisticated, allowing machines to learn complex behaviors and make autonomous decisions in dynamic environments. Machine learning in general is also becoming more accessible to the public and is used in many fields. However, reinforcement learning remains less popular than supervised and unsupervised learning. There are generally fewer classes and courses focusing on reinforcement learning compared to supervised and unsupervised learning. Nonetheless, reinforcement learning is an emerging and promising field with thousands of research paper published every year [1].

At the same time, the spread of smartphones presents a unique opportunity. According to the Global System for Mobile Communications Association, as of 2023, 54% of the world population owns a smartphone [2] and smartphones keep getting better sensors, cameras and computing power. Smartphones also have familiar interfaces and people are used to manipulate them. This makes smartphones accessible and interesting tools in learning or teaching robotics.

Additionally, there already are works aiming to introduce smartphones to robotics such as the work on ROS-Mobile, implementing the Robot Operating System (ROS) to Android [3]. ROS is a widely used framework in robotics research and industry for building robot software. By bringing ROS to mobile platforms, such as smartphones, researchers and developers gain access to a wide range of tools and libraries for developing robotic platforms running with smartphones. Other works in integrating smartphone to robotics to help with education in robotics can date as far as 2014 [4].

While not quite there yet, machine learning training is also making its way to smartphones, using TensorFlow Lite [5] the model can be read by the Android phone. However, concerning training directly on the smartphone, not many libraries exist and the models often require to be hand-made. This means the learning is not as efficient and limited to small models.

1.2 Problem Statement and Objectives

While using a smartphone in robotics can have its advantages, it also comes with challenges. Training Artificial Neural Networks (ANNs) can require a lot of computation and cause latency when giving orders to the agent or during the update. Moreover, making complex calculations for an extended amount of time drains the battery of the phone that needs to be recharged multiple times during training. While computation and battery constraints present challenges, there are also difficulties related to the programming environment.

Java is a powerful and rapid programming language but it is not as used as its counterpart Python for machine learning, especially on mobile apps. It does not share as many libraries and tools to train models or set up RL environments. Additionally, there are limitations when running the program on a smartphone compared to a computer, with regulations and harder to implement external libraries/packages. In addition to these technical challenges, the approach to RL training in the real world environment adds another layer of complexity.

Most projects available online for reinforcement learning use a simulated environment [6], but, while it is safer, ML and RL aim to ultimately be used in real world scenarios. RL training is usually done in simulated environments for various reasons, the most notable ones being safety and time. Indeed, training in simulated environment can be greatly accelerated especially if the environment does not need to be rendered. Moreover, the environment is waiting for the agent's update to update itself which is not the case when training in the real world. Additionally, some systems do not have simulated environments attached able to integrate RL or that can translate perfectly into the real-world environment. Although training directly in the real world poses numerous challenges, it is also the final goal in robotics to have the model work on a real robot. These challenges include safety of exploration during training, the large computation and possible latency resulting from it or the time and manpower required for the training. However, in a real world environment, having to train an agent for 40 hours can be hard to manage. It is also notably more difficult to train in parallel with numerous episodes running simultaneously and updating the same model.

This thesis aims to see the feasibility of implementing RL entirely on an Android smartphone. Moreover, implementing it on an educational platform such as OpenBot to verify the potential for an educational and pratical RL project. The policy gradient algorithm is implemented on the smartphone to train a model to stay on the track of a simple circuit for 15 seconds. The OpenBot platform used in this thesis is described in chapter 4, it is accessible and affordable to build and use. OpenBot has no simulated environment, its learning is done only with real-world interactions.

In this study, the smartphone used is a Samsung Galaxy S22 Ultra with for its GPU the Samsung Xclipse 920 and the Octa-core CPU. The computer used has the NVIDIA GeForce RTX 4070 Ti for GPU or the 13th Gen Intel(R) Core(TM) i7-13700F as CPU. Android Studio is the software used to program the OpenBot app.

The execution of this research project relied heavily on the insights offered in the second edition of *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto [7]. This influential textbook provided a great framework for understanding the principles and methodologies of reinforcement learning.

1.3 Structure of the Manuscript

The remainder of the manuscript is structured as follows:

- Chapters 2 and 3 introduce Reinforcement Learning:
  - The sections 2.1 and 2.2 present the origin, important concepts and keywords in RL
  - The section 2.3 discusses challenges with RL encountered during this thesis
  - The chapter 3 describes different RL algorithms and introduces Artificial Neural Networks
- Chapter 4 presents the robotic platform OpenBot :
  - The section 4.1 describes the hardware components, mechanical assembly and electrical configuration

  - The sections 4.2 and 4.3 introduce the mobile application (App), the python scripts and the python server

- Chapter 5 describes three foundational experiments:
  - A preliminary experiment to implement RL on OpenBot is presented in section 5.1
  - Section 5.2 shows the implementation of the previously introduced RL algorithms in two scenarios: Cartpole in subsection 5.2.1 and Pong in subsection 5.2.2

- Chapter 6 presents the methodology for implementing RL on OpenBot:
  - The experiment and its objective are presented in section 6.1
  - The state, rewards, actions, termination and exploration-exploitation are implemented in section 6.2
  - The section 6.3 describes how the Model is initialized and how policy gradient is implemented
  - The difficulties encountered during the implementation are presented in section 6.4
  - Section 6.5 provides a simple protocol to repeat this experiment

- Safety features are presented for the robot and its surrounding during training in chapter 7

- The results are presented in chapter 8

- chapter 9 discusses the results:
  - The results are interpreted and the limitations of the study discussed in sections 9.1 and 9.2
  - Section 9.3 proposes recommendations for future research or work
  - Section 9.4 presents additional notes concerning the choice of the algorithm and the foundational experiments in

- Chapter 10 is the conclusion of this study

2 Introduction to Reinforcement Learning

This chapter introduces reinforcement learning, starting with its origin. Next, the terms frequently used in RL are listed and Markov Decision Processes (MDPs) are rapidly explained. Then, the Bellman equations at the base of most RL algorithms are presented. The equations are followed with an important part of reinforcement learning: exploration vs. exploitation during training. This allows the model to learn from new experiences during exploration and exploiting what it has learned afterwards. Finally, the challenges of implementing RL directly in the real-world environment and entirely on a smartphone are discussed.

2.1 Origin and Application

Reinforcement learning is a type of machine learning where an agent learns to make decisions by trying different actions and observing the outcomes. The agent aims to maximize the rewards it receives from its actions over time. The idea behind it comes from psychology research such as the one from Thorndike [8] in 1911 in *Animal Intelligence*, that gives the following Law of Effect:

> ”**The Law of Effect is that**: Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.”

Other psychological studies also affirmed and contributed to the beginning of RL such as the work of B.F Skinner [9] [10] in *The Behavior of Organisms* or *Science and Human Behavior* adding to the theory of behavioral learning from rewards and punishment.

But it was only later, in the 50s that RL started to apply to machine learning with the multiple contributions of Richard Bellman. He introduced Dynamic Programming [11], Markov Decision Processes (MDPs) [12] and laid the foundations of Bellman equations commonly used in reinforcement learning and described in subsection 2.2.3. Then, Arthur Samuel worked on one of the first practical application of reinforcement learning using the game of checkers [13]. In his study, he explained how the model is ”looking-ahead” each possible moves and computing positional advantage to select the ”best move” which can be compared to trying different actions and observing the outcome to make decisions based on positive and negative values.

Afterwards, many fundamental algorithm in reinforcement learning such as Q-learning [14], Temporal Difference [15], Sarsa [15], Dyna-Q [16], Deep Q-Learning [17] and Policy Gradient [18] were introduced. Q-learning, Deep Q-learning and Policy Gradient are presented in chapter 3 and used in this study.

Research in reinforcement learning extends across various domains. In robotics, for instance, researchers can focus on training robotic hands to manipulate objects with precision and agility. Notable examples include the work of an OpenAI team focused on reorienting a cube in a desired configuration using an intricate robotic hand [19]. In gaming, RL algorithms have made significant progress, as showed by the work of Mnih et al. who employed the Deep-Q-Network algorithm to train models to play Atari games [17]. Achieving human-level performance or better, these models represent a significant advancement in gaming AI. Then, in healthcare, RL can be used to optimize treatment strategies, with promising results observed in applications such as sepsis treatment [20]. Furthermore, RL techniques have found applications in finance and trading, one example is algorithms aiming to maximize cumulative returns through portfolio management strategies [21]. RL can also be used with autonomous system, the interesting work of Kendall et al. shows the possibility of a real car to learn to stay on the road in a efficient time manner which has a similar objective to the work presented in this study [22]. However, they first used a simulated environment in order to fine tune hyperparameters which is not done in this research and the computation is done on board using a NVIDIA Drive PX2 computer. The NVIDIA Drive PX2 is designed specifically for training autonomous cars. This allows the training of a complex Deep Reinforcement Learning (DRL) model which is currently not feasible on a smartphone.

While the field of reinforcement learning is evolving rapidly, it remains primarily in the experimental and research stages. It has only a few industrial applications, unlike more established machine learning methods such as supervised learning, which already have multiple industrial applications. Although its potential has been demonstrated multiple times, reinforcement learning still faces numerous challenges. These include issues with sampling efficiency, designing robust reward functions, and the increasing dependence of DRL on specific environments [23]. Finally, RL is not taught as much as the other ML methods such as Supervised and Unsupervised Learning making it less accessible for beginners and students despite its potential for solving complex real-world problems.

2.2 First Steps into Reinforcement Learning

This section introduces the foundational concepts of reinforcement learning. It begins with an explanation of key terms and concepts, followed by the presentation of Markov Decision Processes (MDPs) and Bellman equations. Finally, the concept of exploration and exploitation in RL is presented.

2.2.1 Introduction to terms used in Reinforcement Learning

Following is a list of all the important terms in RL and their meaning, they will be used throughout this study and can be referenced back here.

1. Agent: The entity or system that learns and interacts with the environment in RL.
2. Environment: The external system or surroundings with which the agent interacts.
3. State s: A specific situation or configuration of the environment at a particular time. The state at time t is written as S_t.

4. Action a: The decision or choice made by the agent in response to a given state. The action at time t is written as A_t.

5. Reward R: A scalar value that indicates the immediate feedback or outcome received by the agent after taking an action in a specific state.

6. Return G: The return corresponds to the sum of the rewards.

7. Policy π: The strategy or set of rules that governs the agent's decision-making process, mapping states to actions.

8. Value Function $v_\pi(s)$: A function that estimates the expected cumulative reward or value of being in a particular state and following a particular policy.

9. Q-Value (Action-Value Function) on policy $q_\pi(s, a)$: A function that estimates the expected cumulative reward of taking a particular action in a particular state and following a particular policy thereafter.

10. Q-Value (Action-Value Function) off policy $q(s, a)$: A function that estimates the expected cumulative reward of taking a particular action in a particular state, computed using past experience.

11. Off-Policy Learning: A learning method in RL where the agent learns from a different policy than the one it uses to select actions.

12. On-Policy Learning: A learning method in RL where the agent learns from the same policy that it uses to select actions.

Let's take a simple example, a cat (considered here the agent) in its initial position that can choose between two actions: going right or going left. On its left, there is a dog, and on its right, there is a mouse. The cat does not know the consequence of each action and will be moving at random. Since the cat is moving at random, the policy is equiprobable, meaning the probability of taking the action a (left or right) in any state $s \in \mathcal{S}$, $\mathcal{S}$ being all the possibles state in the environment, under the policy is $\pi(a|s) = 0.5$ with a any action in the set of possible action $\mathcal{A}$. This situation is illustrated in the figure 2.1 below. Here, the cat is pleased to find a mouse so going right would provide a positive reward $R = 1$. However, if the cat goes to the left, it will meet the dog and will be displeased, the reward is then set to $R = -1$. In this scenario, let's define the two actions as follows: $[left, right] \in \mathcal{A}$, the action-value function obtained is $q_\pi(S_0, right) = 1$ and $q_\pi(S_0, left) = -1$, S_0 being the initial state. Finally, in this situation, the agent is as likely to go to the left as the right, meaning its value function under the equiprobable policy in the initial state S_0 would be $v_\pi(S_0) = 0.5 * 1 + 0.5 * -1 = 0$.

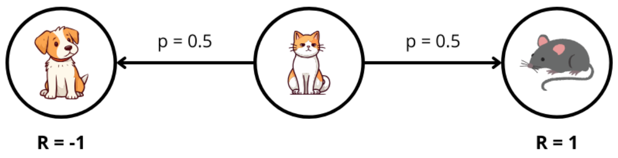


Figure 2.1: Illustration of the Cat (agent) situation and its two possible action.

2.2.2 Markov Decision Processes

A Markov Decision Process (MDP) is the mathematical framework used to model decision-making problems in which an agent interacts with an environment over a series of discrete time steps. MDPs serve as the foundational framework within reinforcement learning. Using the previous scenario with the cat, here is how it would be described as a MDP:

- **States** ($\mathcal{S}$):
  - S_0: Initial position of the cat.
  - S_{dog}: Cat meeting the dog.
  - S_{mouse}: Cat meeting the mouse.
- **Actions** ($\mathcal{A}$):
  - Move left: $left$
  - Move right: $right$
- **Transition Probabilities** (P):
  - Deterministic transitions:

$$\begin{aligned} P(S_{dog}|S_0, left) = 1 \\ P(S_{dog}|S_0, right) = 0 \\ P(S_{mouse}|S_0, right) = 1 \\ P(S_{mouse}|S_0, left) = 0 \end{aligned}$$

- **Rewards** (R):
  - Immediate rewards:

$$\begin{aligned} R(S_0, left) = -1 \\ R(S_0, right) = 1 \end{aligned}$$

- **Policy** (π):
  - Equiprobable policy:

$$\begin{aligned} \pi(left|S_0) = 0.5 \\ \pi(right|S_0) = 0.5 \end{aligned}$$

Describing the scenario as a MDP is helpful when programming the algorithm and understanding how to use the Bellman equations explained just below.

2.2.3 Bellman equations

Previously, a simple scenario was presented, however, the environment is not usually as simple, with more actions and states available. The value function v under the policy π was simple to compute, but rapidly becomes more complex when adding steps and possible actions.

Following is the general equation for the value function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.1}$$

The value $v_\pi(s)$ corresponds to the expected cumulative reward also called return that the agent gets from being in a state $s \in \mathcal{S}$ and following a policy π. The first sum $\sum\limits_a \pi(a|s)$ indicates the probability of taking each action a in state s under the policy π. The second sum $\sum\limits_{s',r} p(s', r|s, a)$ summing over all possible outcomes of the action a in state s, weighted by their probabilities $p(s', r|s, a)$, in here s' is the next state and r the immediate reward for reaching this new state. The last expression $[r + \gamma v_\pi(s')]$ calculates the immediate reward r received upon transitioning to state s' from state s after taking action a and the discounted value $\gamma v_\pi(s')$ of the next state under the same policy. The discount factor γ is a tunable hyper-parameter that ensures immediate reward are valued more than future rewards.

Another important function is the value-action function $q_\pi(s, a)$ corresponding to the expected cumulative reward that the agent gets from being in a state s and then taking the specific action a. It is similar to the value function, except the action is already selected, the first sum is removed and the value-action function is obtained:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.2}$$

The two equations 2.1 and 2.2 are the "Bellman Expectation Equation", they compute the expected returns from a state or state-action pair following a specific policy π. Now, the aim of reinforcement learning is to optimize the return. For it, there are the "Bellman Optimality Equations", they characterize the optimal policy and compute the optimal value of the state and state-action pairs. They use the $\max$ operation function to find the optimal action, following are the Bellman Optimality Equations:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_*(s, a) \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.3}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.4}$$

The $*$ represents the optimal policy (with the largest return), so the equation 2.3 corresponds to the optimal value function $v_*(s)$ for state s. It is the maximum expected return that can be achieved from state s by following the optimal policy. It indicates that the value of state s is equal to the largest expected return obtained by taking the maximum over all possible actions a and summing over all possible next states s' and immediate rewards r that can occur when taking action a in state s.

The second equation 2.4 corresponds to the optimal action-value function for state-action pair (s, a). It represents the maximum expected return that can be achieved by taking action a in state s under the optimal policy. It sums over all possible next states s and immediate rewards r

that can occur when taking action a in state s, weighted by their probabilities $p(s', r|s, a)$. Each term in the sum represents the immediate reward r obtained upon transitioning to state s' from state s by taking action a, plus the discounted maximum value $\gamma \max_{a'} q_*(s', a')$ of the next state s' under the optimal policy $*$. Once again, the discount factor γ is a tunable hyper-parameter that ensures immediate reward are valued more than future rewards.

More information about the Bellman's equation can be found in the Chapter 3 of the second edition of *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto [7].

2.2.4 Exploration vs Exploitation in Reinforcement Learning

In RL, exploitation involves selecting the action with the highest expected return according to the current policy, while exploration is trying alternative actions to discover potentially better strategies. Exploration often involves random selection of actions and significant research is devoted to finding optimal strategies for balancing exploration and exploitation effectively. Additionally, considerable effort is directed toward optimizing exploration strategies since exploring randomly can be very time consuming.

In this study, the epsilon-greedy method is employed as the exploration strategy. Its origin is difficult to trace but was made popular by Richard S. Sutton and Andrew G. Barto in the first edition of their book published in 1998 [7]. The epsilon-greedy method is simple but is still a very popular approach for balancing exploration and exploitation. Here's how it works:

At the start of the learning process, the value of epsilon (ϵ) is initialized. If it is initialized to a value close to 1, it indicates that the agent prioritizes exploration over exploitation. As the learning progresses, epsilon gradually decreases over time according to a predefined decay rate. During each iteration or time step, the agent faces a decision: whether to explore or exploit. To make this decision, the agent generates a random number between 0 and 1. If this random number is less than epsilon, the agent chooses to explore by selecting a random action from the action space. However, if the random number exceeds epsilon, the agent follows its current policy and exploits the action with the highest estimated value. By gradually reducing epsilon over time, the agent's behavior shifts from exploration towards exploitation as it gains more experience and confidence in its learned policy.

In Figure 2.2, the evolution of epsilon is illustrated with an initial epsilon value of 0.15 and a decay rate of 0.999 over 10000 episodes. In the first few episodes, there should be a bit more than 1 in 10 actions that are exploratory (random) and near 6000 episodes the exploration is over.

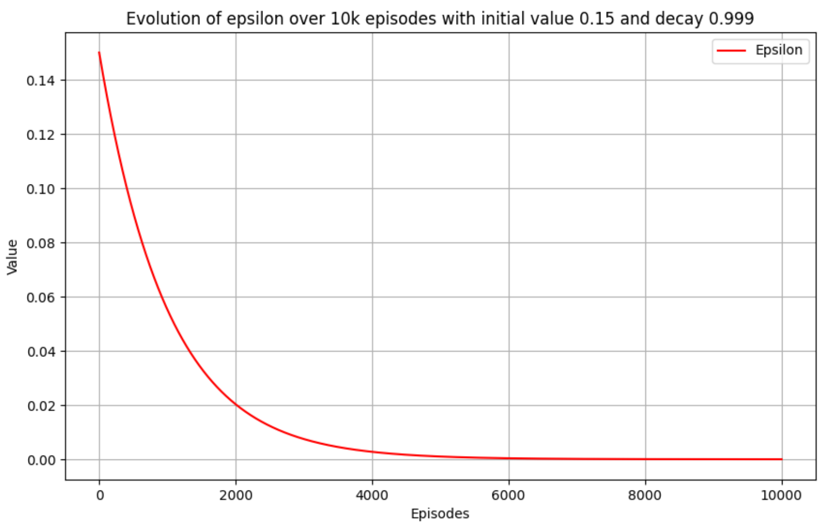


Figure 2.2: Graphical representation of epsilon evolution over 10000 episodes with initial ϵ set to 0.15 and decay rate to 0.999.

2.3 Challenges with Reinforcement Learning

The section below addresses the difficulties encountered when applying reinforcement learning in real-world scenarios and on mobile devices.

2.3.1 Challenges of Implementing Reinforcement Learning on Hardware-Constrained Platforms

There are two main challenges of implementing RL on a platform without simulated environments (Hardware-Constrained) discussed in this study: Safety and time consumption coming with training a model.

Training an agent in the real world comes with an obvious safety concern, as any action taken by the agent have real-world consequences. Potentially the agent could cause harm to itself, its surrounding or even someone. Fortunately, the platform used in this study and presented later in chapter 4 is relatively robust. Most pieces are replaceable, moreover, they are unlikely to break because of a crash even at the maximum speed. However, it is important to stay careful of stairs or steps since the smartphone is the most valuable item that can be damaged if the robot falls badly. The most likely pieces to break are the push sensors in the front and the back of the robot but they are easily replaced or disconnected. Nevertheless, it is important to implement safety features during training to protect the robot, its surrounding and people or pets that may be present during training. The implementation of two safety features are discussed in chapter 7 to avoid the robot straying and putting itself or others in danger.

Finally, training an agent using reinforcement learning can be very time-consuming. Indeed, it can take tens of thousands of episodes for the agent to reach an optimal policy, and even without rendering the environment and accelerating the process, it can take days to train, as seen in teaching an agent to play Pong in the section 5.2.2. In the real-world environment, accelerating the process by skipping frames or not rendering is not a possibility. Having multiple agents running in parallel to train the same model, while maybe conceivable using a single main computing unit training the model with all the agent reporting to it, would require a person per agent to monitor and can become very expensive.

In this study, only one agent is available and one person to monitor it. To avoid taking too long making a model from scratch and dealing with trial and errors in the real environment, first trials were made on two scenario : Cartpole presented in section 5.2.1 and Pong presented in section 5.2.2.

2.3.2 Challenges of Implementing Reinforcement Learning on a Smartphone

There are multiple challenges in implementing RL on a smartphone, such as the computing power of smartphones compared to traditional desktop or server setups and the battery consumption during learning. While there are efforts to develop machine learning frameworks on smartphones, with TensorFlow Lite [5] making models in a readable format to Android devices for instance, they often lack functionality, performance and the community support that Python has with ML.

The computing power difference can cause latency when computing the forward pass of the model, especially in deep ANNs that can end up with millions of parameters. This also significantly drains the battery, can heat up the smartphone and is also due to the absence of optimization for ML in smartphones. The model has to be shallow and the task simplified in order to achieve a successful RL implementation. Otherwise, the latency caused by the computation during the forward pass is a concern for an agent interacting in the real world. The external environment is not waiting for the agent to update to update itself. Moreover, by the time the action is sent to the agent, there is the possibility that the agent is in a different position and will crash.

3 Reinforcement Learning Methods and Algorithms Studied in this Thesis

In this chapter, the different algorithms used and discussed in this study are introduced. This is a simple introduction to these algorithms and their applications are discussed later in Chapters 5 and 6. Artificial Neural Networks (ANNs) are also introduced since Deep Q-Learning and Policy Gradient use them. To learn more about these algorithms, reading the Chapters 5, 6, 9 and 13 of the book *Reinforcement Learning: An Introduction* [7] is highly recommended since they were used as a base for this research.

3.1 Q-Learning

Q-Learning is considered one of the early breakthroughs of reinforcement learning, it was introduced by Christopher Watkins in 1989 [14] in his thesis *Learning from Delayed Rewards* and was detailed in a technical note published in 1992 [24]. Q-learning is considered an *off-policy* control method because it approximates the optimal action-value function q_* without taking into account the policy. This means the agent is learning using its past experience and updating after each new experience.

As a reminder, the action-value function is defined in 2.2.1 as the estimation of the expected cumulative reward of taking a specific action in a particular state. It is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{3.1}$$

The action-value function Q at state S_t and opting for the action A_t at time t is updated after observing the new state S_{t+1} and obtained reward R_{t+1} after taking that action. For this, Q needs to be arbitrarily initialized for all states $s \in \mathcal{S}$ and all actions $a \in \mathcal{A}$ except for the terminal state that is set to $Q(terminal, \cdot) = 0$.

Instead of following a policy, this equation 3.1 uses $\max_a Q(S_{t+1}, a)$ that returns the largest expected return when in the new state S_{t+1} by comparing the value for all the possible action in that state. It is then multiplied by a discounting factor γ that needs to be determined. The pair S_t, A_t is updated with a step size α, the obtained reward and the value of the best state-action pair in the new state S_{t+1} after taking the action A_t.

Then, after visiting the states enough times and selecting various action, the action-value function should reach optimality. The algorithm below for Q-learning is quite straightforward and should reach optimality as long as all pair continue to be updated, this is proved in the book *Reinforcement Learning: An Introduction* [7] in Chapter 5.

Algorithm 1 Q-Learning with ϵ-greedy

```
Parameters: step size $\alpha \in (0, 1]$, small epsilon for $\epsilon$-greedy $\epsilon > 0$, discounting factor $\gamma$
Initialization: Initialize arbitrarily $Q(s, a)$ for all $s \in \mathcal{S}^+$ and all $a \in \mathcal{A}$. Initialize the terminal state $Q(s_{terminal}, \cdot) = 0$
for all Episodes do
    Initialize $\mathcal{S}$
    for all Steps of Episode do
        Choose action $\mathcal{A}$ from $\mathcal{S}$ using $\epsilon$-greedy
        Take action $\mathcal{A}$ and observe resulting state $\mathcal{S}'$ and reward $\mathcal{R}$
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
        $\mathcal{S} \leftarrow \mathcal{S}'$
    end for
end for
```

3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs), as the name implies, are models inspired by the structure and function of biological neural networks in the brain. One of the first work introducing ANNs is from Warren S. McCulloh and Walter Pits in 1943 [25]. ANNs consist of interconnected nodes (or neurons) organized into layers: an input layer and output layer, with hidden layers in-between. The model can be $shallow$ with only one hidden layer or $deep$ with multiple hidden layers. Each node of each layer is connected to all the node of the following layer with different weights. In addition to these connections, each node applies an activation function to its input before passing it to the next layer. This activation function introduces non-linearity into the network, enabling it to capture complex relationships in the data. This allows the model to find $features$ in the input leading to the final output. A common example is a model taking as input pictures and giving as output whether the image is a dog or a cat as seen in figure 3.1. The features extracted by the model can be focusing on colors, angles and are not easily understood by humans when the model is deep and complex. However, image recognition and labeling has become more and more accurate over the years thanks to Convolutional Neural Networks and other methods but they are not discussed in this study.

ANNs are mostly used for non-linear function approximation, meaning for cases where the relationship between variables are not linear which is the most common scenario. This non linearity is achieved thanks to the activation function. The activation function is applied to the weighted sum z at a node, the most common ones are:

- Sigmoid function:
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

- Hyperbolic Tangent (tanh) function:
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.3}$$

- Rectified Linear Unit (ReLU) function:
$$\text{ReLU}(z) = \max(0, z) \tag{3.4}$$

- Softmax function:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{3.5}$$

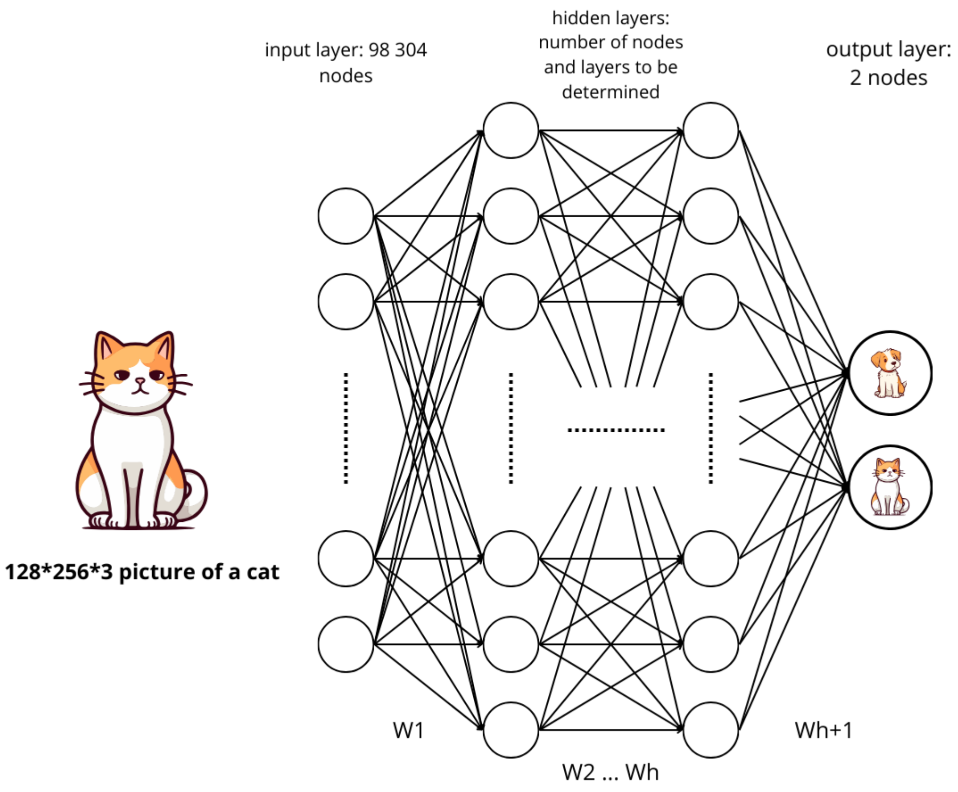

input layer: 98 304 nodes
hidden layers: number of nodes and layers to be determined
output layer: 2 nodes
128*256*3 picture of a cat
W1
W2 ... Wh
Wh+1

Figure 3.1: Simplified Artificial Neural Network with a picture of a cat of size 128x256x3 as the input, followed by an undetermined number of hidden layers and nodes and finally the output layer with two nodes; probability of the image to be a dog and probability to be a cat. In between each layer, there are Weights connecting each node to all the following nodes of the next layer.

In order to have good results, the model requires $training$, before starting the training, the weights connecting the nodes are initialized. There are different ways for initialization, the $random$ initialization with a small range, the $normal$ or $naive$ initialization where the values are drawn from a normal distribution with mean 0 and small standard deviation. Finally, in this study is also used the $Xavier$ or $Glorot$ initialization with the weights initialized using the first equation of their study [26]:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{3.6}$$

Where $U\left[-a, a\right]$ is the uniform distribution in the interval $(-a, a)$ and n is the size of the previous layer (the number of columns of W).

The objective of ANNs is to have the model $learn$ or discover features to give the most accurate output. During training, the weights need to be updated, in order to have better results. In supervised learning, the model has labeled data known as $ground\ truth$ to which it can compare how far away from the truth it was when it was given the input and provided the output. In reinforcement learning, there are no ground truth to compare the results, what is given is whether the action taken had a positive, negative or neutral impact to reach the given objective compared to the expected outcome. In supervised learning, the discrepancy between predicted and actual values is typically quantified using a $loss\ function$, while in reinforcement learning, the assessment of the agent's actions against the desired objective is similar to the concept of a loss function.

The action of giving the input to the model that will then compute the output is called the $forward\ pass$, the weights are not updated during this forward pass. After the forward pass, the accuracy of the prediction is computed in order to update the weights. The updating of the weights is done after what is called the $backward\ pass$. During this backpropagation, the gradient of the loss function with respect to each weight in the network is calculated, which indicates how much each weight contributed to the error. Then, using this gradient information, the model updates its weights to minimize the error, improving its performance over time.

ANNs have become increasingly powerful and are now commonly used in everyday applications. They greatly helped advance the field of reinforcement learning, particularly with algorithms like deep Q-learning and policy gradient, which are presented below.

3.3 Deep Q-Learning

Deep Q-Learning (DQL) was introduced in 2013 by DeepMind to play Atari games [17], it is a mix between Q-Learning discussed in the section 3.1 and the Artificial Neural Networks in the section 3.2. Many Atari games have images as input with other parameters such as life points or different bonuses. It is impossible to compute all possible states if the image is coloured and is even as small as $32 * 32 * 3$ pixels, there are 256^{3072} different combinations. This is why the use of ANNs is so powerful, if done properly, it can find the important features of the input in order to provide an accurate output.

In DQL, the ANN is used to compute the action-value function $Q(S, a)$ for each action $a \in \mathcal{A}$ at a particular state S. The output size is then the size of the number of action. Here is the algorithm for Deep-Q Learning inspired by the study [17]:

Algorithm 2 Deep Q-Learning with ϵ-greedy and Experience Replay

```
**Parameters:** small epsilon for $\epsilon$-greedy $\epsilon > 0$, discounting factor $\gamma$
**Initialization:** Initialize replay memory $\mathcal{D}$ to capacity $\mathcal{N}$
Initialize action-value function $Q$ with random weights $\theta$
for all Episodes do
    Initialize state $s_1 = x_1$ and preprocessed state $\phi_1 = \phi(s_1)$
    for all Steps of Episode do
        Select random action with probability $\epsilon$
        Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Take action $a_t$ and observe resulting image $x_{t+1}$ and reward $r_t$
        Set $s_{t+1} = s_t$, $a_t$, $x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) in $\mathcal{D}$
        Sample random minibatch of transitions ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) from $\mathcal{D}$
        Set $y = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

In this algorithm, x corresponds to the image given by the Atari game, it is preprocessed using the function ϕ so useless features are already removed to help with computation and time required for learning. To reduce computation, random minibatch from the memory are taken to train the model, meaning the order is not important and the state-action pair are randomly 'visited' by the model.

Using this method, they were able to play 7 Atari games. Their approach surpassed previous methods in 6 of these games and even outperformed human experts in 3 of them. In this study, the method is used in playing Cartpole in section 5.2.1 and completed its training in less than 600 episodes.

3.4 Policy Gradient

Policy gradient is another interesting reinforcement learning method. Unlike Q-Learning and DQN, in policy gradient methods, the aim is to improve the policy using gradients to update the ANN model during backpropagation. The aim is to maximize the rewards when updating the model, using gradient ascent. These type of PG methods were introduced in 1992 and were called REINFORCE algorithms [18]. In this case, the PG learns a $parametrized\ policy$ that can select an action without using a value function. A value function can still be interesting to learn the policy parameter but is not required during action selection. As the objective is to maximize the performance, the update of the model is done using gradient $ascent$ using a scalar performance measure $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha\Delta\widehat{J(\theta_t)} \tag{3.7}$$

Here, θ represents the parameters of the policy, in this case the weights of the model, and $J(\theta)$ is a performance measure indicating how well the policy performs. By iteratively adjusting θ in the direction of increasing $J(\theta)$, the policy improves over time, leading to better performance in the given task.

Parameterized policies (ANNs) offer flexibility and can adapt to different environments without explicitly computing value estimates for each action. This approach simplifies the learning process and allows for more direct optimization of the policy. PGs can be implemented in various ways and depends on the situation. In this study, the policy gradient algorithm is used when updating the weights of the ANN model. The Root Mean Square Propagation (RMSProp) is the performance measure $\Delta\widehat{J(\theta_t)}$ and is presented in subsection 5.2.2 with the equation 5.9 .

4 Presentation of OpenBot

This chapter introduces OpenBot, an accessible and innovative platform for real-world experimentation. OpenBot's hardware, mechanical assembly and electrical configuration are presented. Then, the app used to control Openbot is introduced as well as the connected server and python codes.

4.1 Description of OpenBot

OpenBot is a small robotic platform seen in Figure 4.1 made by Matthias Müller and Vladlen Koltun in 2020 [27]. It is an accessible platform with open-source code. The small robotic car has a smartphone as its main computing unit, it is used for control and various sensors. This robot was designed to be cost efficient and use supervised learning algorithms to solve different tasks such as people following, object tracking or point goal navigation. The code is well documented on their GitHub [28] and provides explanation about training one's own model for a specific task.

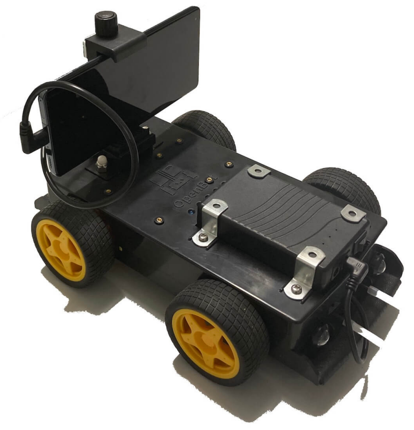


Figure 4.1: Picture of OpenBot Ready-To-Run model with smartphone mounted. From OpenBot's GitHub [28].

4.1.1 Hardware Components

The OpenBot platform has the following key hardware components:

- A standard RC car chassis, either bought or 3D printed.
- Four DC motors with tires.

- An Android smartphone for Control and Computation
- Arduino Nano
- Custom PCBs
- A USB on-the-go (OTG) cable for communication between the smartphone and the microcontroller.
- Various sensors, including the smartphone's buit-in camera for vision-based navigation and accelerometer, ultrasonic or push buttons for obstacle detection.
- Another phone or controller (PS4, Xbox,...) is required to send controls to the OpenBot.

For more information on how to build your robot, where to order it or about the custom PCBs, please refer to the OpenBot paper [27] and GitHub repository [28]. For this thesis, the OpenBot was the Ready-To-Run (RTR) model seen in Figure 4.1.

4.1.2 Mechanical Assembly & Electrical Configuration

OpenBot is designed to be compact, housing its microcontroller, motors, and PCBs within the 3D printed mount. It has an adjustable smartphone mount suitable for various sizes. While the RTR model is chargeable, the Do It Yourself (DIY) version operates using three 18650 batteries.

Figure 4.2 shows the configuration with four motors and the various components enclosed within the mount. The paired motors ensure synchronous control of the right and left wheels. When a single motor stops functionning, the paired motor also ceases to operate despite receiving the command.

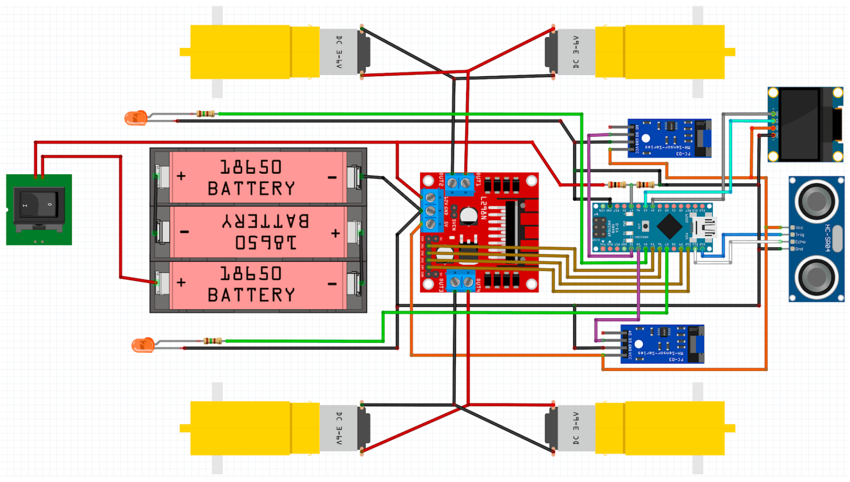


Figure 4.2: Wiring diagram. Electrical connections between battery, motor driver, microcontroller, speed sensors, sonar sensor and indicator LEDs. From the OpenBot GitHub and paper [27] [28].

4.2 Presentation of the App

OpenBot uses a simple Android app for control and different feature, with the following being the most pertinent for this study:

- **Autopilot**: This feature uses a trained tflite model uploaded to the smartphone, using data from the phone such as the camera, accelerometer, gps and more as inputs. In a prior project [29], this feature was successfully employed for autonomous navigation around a predefined circuit.
- **Model Management**: This is an interface for managing all uploaded models on the smartphone. Users can organize, delete, or download models, categorizing them as needed.
- **Data Collection**: A critical aspect of this experiment, the Data Collection feature records data from various sensors and the control log, saving the information in `.txt` or `.png` formats. This data is then uploaded to the smartphone or a remote Python server for machine learning algorithm training.
- **Free Roam**: This feature enables users to control the robot either through a controller or another Android phone equipped with the controller app, available on the OpenBot GitHub repository [28].

4.3 Introduction to the Server and Python Scripts

As mentioned earlier, smartphones currently lack the capability to train complex machine learning models directly on-device due to the absence of libraries and limited computational power compared to common computers. To address this limitation, OpenBot uses a Python server used to exchange data between the smartphone and a computer. This allows the use of Python scripts for model training with TensorFlow [5].

4.3.1 Python Server

The Python server takes care of the flow of data between the robot and the computer. The process involves the robot transmitting compiled data to the server. The folder containing the datasets is stored in the designated 'uploaded' directory. This folder features a preview option seen in fig 4.3 and the users can relocate it to either the training or testing directory, or delete it in case of issues.

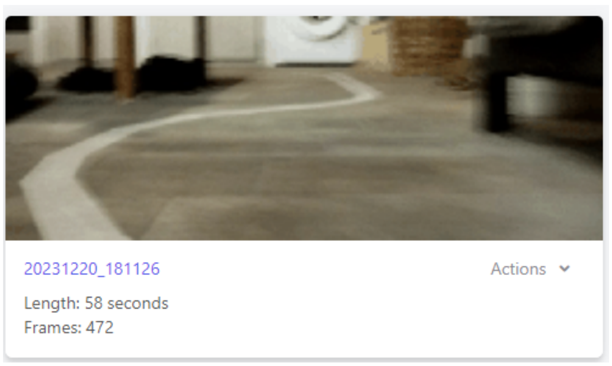
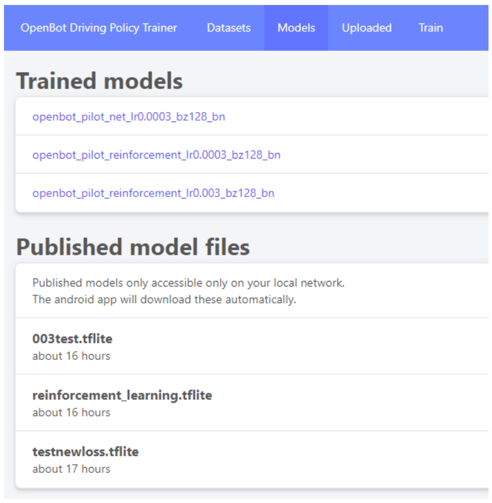


Figure 4.3: Preview of the data collected with pictures flashed consecutively like a video, the option to move directory is also displayed as 'Actions'.

After training, the server recognizes the model in tflite format, this can be seen in figure 4.4. This registered model, customizable with a user-specified name, can be uploaded to the

OpenBot app as seen in figure 4.5. It is important to note that assigning an existing model name will replace the older model.



Trained models

openbot_pilot_net_lr0.0003_bz128_bn

openbot_pilot_reinforcement_lr0.0003_bz128_bn

openbot_pilot_reinforcement_lr0.003_bz128_bn

Published model files

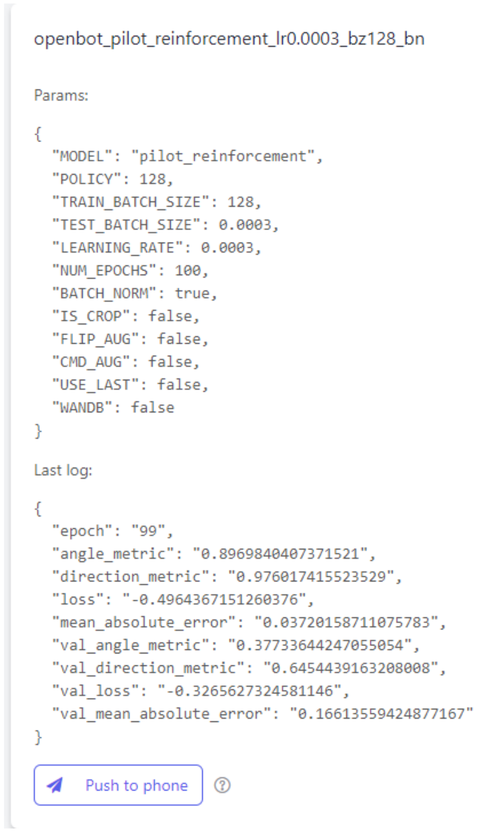
Published models only accessible only on your local network.
The android app will download these automatically.

003test.tflite
about 16 hours

reinforcement_learning.tflite
about 16 hours

testnewloss.tflite
about 17 hours

Figure 4.4: Screenshot of the server's model uploading scheme. The same model can be uploaded multiple times under different names and uploaded models can be deleted.



openbot_pilot_reinforcement_lr0.0003_bz128_bn

Params:

```
{
  "MODEL": "pilot_reinforcement",
  "POLICY": 128,
  "TRAIN_BATCH_SIZE": 128,
  "TEST_BATCH_SIZE": 0.0003,
  "LEARNING_RATE": 0.0003,
  "NUM_EPOCHS": 100,
  "BATCH_NORM": true,
  "IS_CROP": false,
  "FLIP_AUG": false,
  "CMD_AUG": false,
  "USE_LAST": false,
  "WANDB": false
}
```

Last log:

```
{
  "epoch": "99",
  "angle_metric": "0.8969840407371521",
  "direction_metric": "0.976017415523529",
  "loss": "-0.4964367151260376",
  "mean_absolute_error": "0.03720158711075783",
  "val_angle_metric": "0.37733644247055054",
  "val_direction_metric": "0.6454439163208008",
  "val_loss": "-0.3265627324581146",
  "val_mean_absolute_error": "0.16613559424877167"
}
```

Figure 4.5: Screenshot of the models details, the model is uplaoded to the OpenBot app by clicking the 'Push to Phone' button.

4.3.2 Python Scripts

There are 12 different python scripts to process the data and train the models. There are also 10 scripts for the python server used for transmitting data from OpenBot to the computer, however, they were not modified because of a lack of knowledge in Python servers programming. Finally, there is a notebook in the .ipynb format for running all the scripts and training the models.

Here is a list of the 12 python scripts and what they do:

- **__init__:** This script takes care of the project structure and verifies the existence of essential directories for storing data and models.
- **associate_frames:** The script is for synchronization between frames and control signals as well as frames and rewards, for further analysis and training of machine learning models.
- **callbacks:** They are used for optimizing and monitoring the training of neural network models. They can be employed based on specific requirements during the training process.
- **data_augmentation:** This script can be used to add diversity to the training dataset, which can improve the generalization and robustness of machine learning models trained on the data. However, it is not adapted to take into account rewards.
- **dataloader:** This class is designed access the labeled data for training neural network models.
- **losses:** This script regroups all the loss functions.
- **metrics:** These metrics are designed to evaluate the performance of models in predicting angles, specifically focusing on the absolute difference and the correct direction of the angles.
- **models:** This script regroups all the available models for training.
- **tfrecord:** The script is used to generate TensorFlow Record (TFRecord) files from a collected dataset. These TFRecord files are a more robust and efficient way of loading data for training the model.
- **tfrecord_utils:** This script is used for manipulating TensorFlow records (TFRecords).
- **train:** The train script is a training script for a neural network using TensorFlow. It includes components for handling data in both directory format and TFRecord format, as well as training and evaluation procedures. The script can be configured to train different models (specified in the –model argument) with various hyperparameters. Additionally, it supports loading data either from directories or TFRecord files, and it provides an option to create TFRecord files from the raw data.
- **utils:** This script includes utility functions for working with TensorFlow and some related libraries. These utilities cover a range of tasks related to machine learning and deep learning workflows.

5 Foundational Experiments

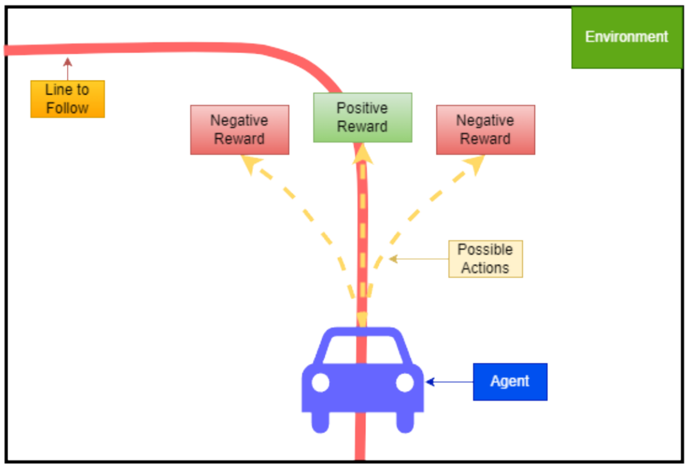
This chapter presents three foundational experiments for this thesis. First, an attempt at implementing reinforcement learning to the OpenBot platform using the python server and scripts was done. This attempt allowed a better understanding of the OpenBot app, codes, how to make a reward function and what are the next steps to implement RL directly on the smartphone. Then, deep Q-learning is implemented to a simple scenario: Cartpole, introducing a first demonstration of a successful RL algorithm. Finally, policy gradient is used to train an agent to play Pong, showing a second successful RL algorithm. All these experiments serve as a strong base for the final implementation on the smartphone.

5.1 Preliminary Attempt at Implementing Reinforcement Learning on OpenBot

This section presents a preliminary attempt at implementing RL on OpenBot and the writing reflects what was believed at the time. This attempt used rewards, added them to the loss function in order to maximize them, following the main idea of the 'Law of Effect' behind RL. However, it lacked the basic RL format, it was not following a MDP nor using a specific RL algorithm. These differences are discussed in the Conclusion and Discussion in the subsection 5.1.7. While this preliminary attempt had a flawed foundation, it was still crucial for this study as it was also an introduction to the methodology for modifying and working with the OpenBot platform and get more familiar with the app, programming in Java and creating a reward function.

5.1.1 Experiment Description and Objectives

The first experiment is a simple line following scenario described in figure 5.1. The main objective is to have the agent learn to do laps on the circuit also seen in figure 5.1 without straying too far away from the line. For this objective, a new feature on the app needs to be created with a reward function. The python code also needs to be modified to take into account the new reward and add it to the update of the model in order to maximize it. The ANN model used is the same as the one used in Supervised Learning for OpenBot and the two methods are compared to make laps around the circuit.



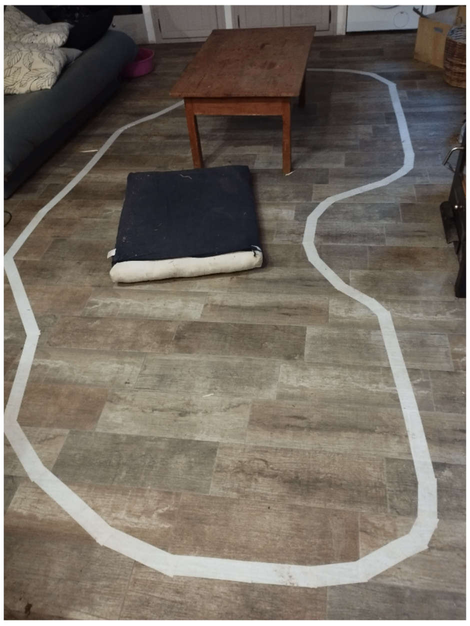


Figure 5.1: Illustration of the reinforcement learning process in the line following scenario on the left and picture of the circuit on the right.

5.1.2 Adding the OpenCV library to Android Studio and OpenBot's Base Code

OpenCV [30] is a well-known open-source computer vision and machine learning software library used for its vast array of image processing capabilities. It offers many functions for image and video analysis, including feature detection, object recognition, and motion tracking. It has extensive documentation and an active community support to help with its use.

The integration of the OpenCV library within Android Studio can be difficult, often requiring a good understanding of versioning intricacies across OpenCV, Android Studio, and the custom code. This process requires careful attention to version compatibility to avoid conflicts and ensure a correct integration.

The integration of the OpenCV library was first done in a simple code environment before being applied to the OpenBot app codebase. This preliminary step helped gain a better understanding of the installation process, which proved to be challenging. Ensuring compatibility with the correct versions required updates to both the Java programming language and the Gradle build system from the OpenBot app.

The details for installing OpenCV to the OpenBot app is given in the appendices in section 10.2.

5.1.3 Creating the new Feature

How to Add a New Feature to the OpenBot App

Following are the steps to incorporate a new feature into the OpenBot app:

1. **Creating a New Fragment and Layout File:** Create a new Fragment along with its corresponding layout file to provide the necessary User Interface (UI) components for the

new feature. The naming convention is $ChosenNameFragment.java$ for the java file and $fragment\_chosen\_name.xml$ for the UI.

2. **Extending the Relevant Class:** Depending on the requirement for the camera preview, extend to your class either the `ControlsFragment.java` or the `CameraFragment.java` class to ensure seamless integration of the new feature within the app's framework.

3. **Update in FeatureList.java:** Add the new feature to the `FeatureList.java` file, specifying its category, subcategory, title, icon, and color. An example of this update is shown below:

```java
ArrayList<SubCategory> subCategories = new ArrayList<>();
...
subCategories.add(new SubCategory(POLICY_GRADIENT,
↪  R.drawable.rtr_tt, "#7268A6"));
...
categories.add(new Category(ALL, subCategories));
```

4. **Update in nav_graph.xml:** Add the fragment within the `nav_graph.xml` file and linked to the `mainFragment` to ensure the navigation within the app.

5. **Navigation Integration:** Inside the `switch` block in the `onItemClick` method of `MainFragment.java`, add the new feature title as a new case. The navigation to the screen is facilitated using its specific action ID, as showed below:

```java
 Navigation.findNavController(requireView())
    .navigate(R.id.action_mainFragment_to_AIFragment);
```

By following these steps, one can integrate a new feature into the OpenBot app, adding their own algorithm to do a specific task.

5.1.4 Implementation of the Reward:

The Reward Function in the Line Following Scenario

In this case, the goal is to have the robot follow a line marked on the ground. A straightforward approach involves a basic reward function, assigning positive numerical values when the robot remains on the line and negative values when it deviates. However, this simplistic setup poses a challenge: what if the robot learns that staying stationary yields the most rewards? To counter this, the system can either assign a positive reward when the robot is in motion or impose negative rewards when no command is given. Another alternative is manual rewarding, where a supervisor can press a button on the controller to administer positive or negative feedback during training, as deemed appropriate.

How the Reward Function was Implemented

Due to limited experience in JavaScript programming, the decision was made to develop and fine-tune the reward function using OpenCV in Python, analyzing images captured at different times of the day. The outlines of the detected line and its centroid were obtained. The centroid calculation is simple, it is the mean of all the contour points' X position.

Following this, a simple Android application was created using Android Studio, where OpenCV was integrated as a library. The interface was designed to display the line contours and centroid, alongside the centroid's distance from the expected central position. Once this code worked properly, it was incorporated to the OpenBot app.

The calculated centroid position is then compared to an ideal one. The ideal centroid position was determined by putting the robot at an ideal position at different points of the circuit and taking the mean value of the centroid. This compared distance is used to determine the reward. The reward r can take the following values:

$$r = [-15, -1, 1, 10, 15]$$

With -15 and 15 negative or positive feedback from the supervisor. Reward is set to $+1$ if the computed distance is inferior to 300, it is set to $+10$ if the distance is inferior to 100 and to -1 when the distance is larger than 300. These value were determined through trials and errors.

On the upside, the computation for this reward function is lightweight and integrates with the standard data logging process without issues. The calculation is performed directly on the smartphone and is transmitted alongside other sensor data, including images, command value logs, and accelerometer readings.

However, there are downsides to consider. First, the reward function's applicability is limited to the specific circuit. Additionally, its sensitivity to variations in light sources, time of day, and weather conditions can lead to inconsistent performance as seen in figure 5.2. To avoid detecting white from objects on the wall or furniture, only the bottom 20% of the picture is taken into account. There are instances where the function fails to detect the line even when it is clearly visible, which can be problematic during the model training phase. And a bad reward function can potentially hinder the learning process and affect the model's performance.

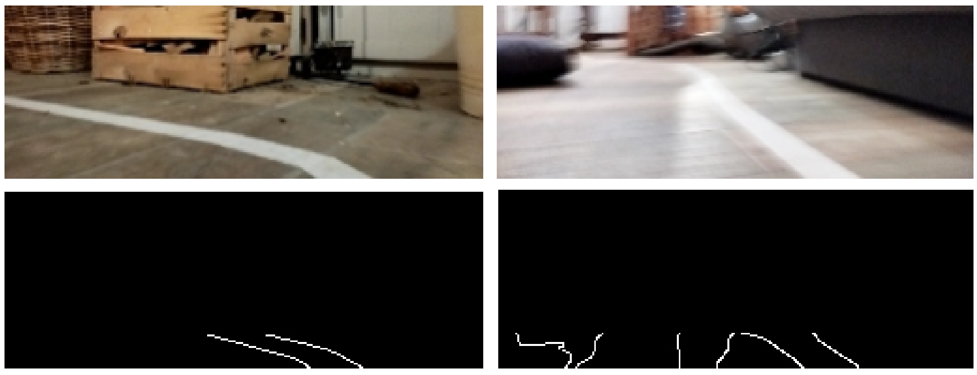


Figure 5.2: Display of photos before and after processing to detect the line to follow. On the left is displayed the ideal result of line detection. On the right is example on how the luminosity impacts the result and could be problematic.

The reward function was implemented directly in the Logger Fragment of the OpenBot app, which means it was calculated while the model is running and the data is collected. In the $processFrame$ function, the OpenCV processing is applied to the same bitmap (image) as the frame collected and the processed image is then saved to a new folder. These images are just to ensure the processing goes as planned and are not used in the model.

Here is the detail of how the image is processed using OpenCV:

1. Conversion to grayscale
2. Addition of contrast to the image with a factor of 1.2.
3. Normalization of the image values to a range between 0 and 255 after contrast adjustment.
4. Detection of white pixels using trial-and-error determined lower and upper bounds of $[200, 170, 170]$ and $[254, 254, 254]$, respectively.
5. Application of median blur with a factor of 9 to the image.
6. Utilization of the $Canny$ function from OpenCV in JavaScript to obtain edges.
7. Retention of only the Region of Interest, specifically the bottom 20% of the image.
8. Calculation of the centroid, with only the x-value considered in computing the reward function.

The processed images are showed in figure 5.2.

Following the code from the Autopilot Fragment and pre-existing code in the Logger Fragment, adding the necessary functions such as $startAutonomousDriving$, $stopAutonomousDriving$, $handleDriveCommandAutonomous$ and $processFrameForAutonomous$, are pretty straightforward. There are also the functions to change the reward to +15 or -15 using the command inputs from the controller and the function changing the reward using the distance between the calculated centroid position and the ideal centroid position. Refer to the GitHub repository [31] for more details on the code.

Presentation of the Model and Algorithm

After implementing the reward to the OpenBot app, it is added to the other sensors like accelerometer, control Logs, etc that are sent to the Python Server. The python scripts need to be modified to retrieve the data concerning the rewards and add them to the loss function so the model tries to maximize them.

In this scenario y corresponds to the command sent to the robot's motors, going from 0 to 255 but is divided by 255 during computation, it takes this form: $(motor_{left}, motor_{right})$. The input to the complex model is the 256*96 picture that goes through 5 convolutional layers before adding the controls y, using Concatenate and Dense until the output of size 2 for the future command sent to OpenBot. The model also uses batch-normalization and dropout. The model is better described in the OpenBot paper [27]. All the modifications made to the scripts are available on the remote repository [31].

Adding the Reward to the Pre-Existing Loss Function

In RL, the aim of the model is to maximize the reward, so in this preliminary experiment, the loss function was modified to take into account the reward and try to maximize it. The determination of the loss function was made through trial and error. Initially, the cumulative sum of rewards and the Mean Squared Error (MSE) 5.2 between y_{true} and y_{pred} were employed.

However, difficulties appeared as the sum of rewards increased with the addition of new data, particularly when striving for unbalanced rewards, predominantly positive. This complexity in weighting led to difficulties and resulted in the poor training of models.

One solution consisted on transitioning to the mean of the rewards (MoR) 5.3, with the objective of maximizing it, noting that it would never surpass 15, the maximum reward possible. Yet, a new challenge emerged. Minimizing the raw difference between prediction and true values resulted in models that overfit and overly generalized decisions.

The supervised learning model implemented in OpenBot adopted a different strategy to compute the loss, focusing on the angle 5.1 rather than the raw difference between values given to the right and left motors. By subtracting the command from the left motor from the command from the right motor, a negative value indicated a right turn, and vice versa. Additionally, this approach approximated the sharpness of the turn, with higher absolute values suggesting a sharper turn. This ensured that, even with variations in the robot's speed, the turn remained consistent with expectations.

Replacing the MSE of the raw values with the MSE of the angle value showed promising results. However, in situations where both motors had identical values, the model opted to assign 0 as the value for each motor, as the difference was the same: 0. Finally, the determination of the loss function was accomplished by integrating both raw values and the angle, as outlined in 5.4. The mean of the rewards is subtracted as the expectation is to get the loss to a minimal value while maximizing the rewards. All three have weights that can be tweaked for optimal results, in this experiment, the final weights were $w_{MSE} = 1$ for the MSEs and $w_{MoR} = 0.8$ for the mean of rewards.

$$\text{angle} = motor_{left} - motor_{right} \tag{5.1}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2 \tag{5.2}$$

$$\text{MoR} = \frac{1}{n}(\sum_{i=1}^{n} r_n) \tag{5.3}$$

$$\text{loss\_function} = w_{MSE} \cdot (MSE_{Angle} + MSE_{Raw}) - w_{MoR} \cdot MoR \tag{5.4}$$

5.1.5 Experiment Protocol

This section describes the protocol for both experiment, the new RL implementation and the original Supervised Learning training already existing within the OpenBot app.

Experiment with Reinforcement Learning

The protocol followed to repeat the experiment using RL is shown below. Figure 5.3 seen at the bottom is a simplified flowchart of the protocol.

1. **Setup:** Turn on the python server following the guide in the OpenBot github [28]. Make sure the phone and computer are connected to the same network and the OpenBot app connects to the server. Note: Verify that the phone is not in 'Lock Orientation' mode.

2. **Initial samples:** To speed up the time-consuming process, a small number of correct laps are taken initially. This method helps improve exploration by reducing excessive randomness.

3. **Train first model:** Train a first model with the data, name it 'reinforcement_learning' and upload it to the OpenBot app.

4. **Initiate Exploration:** Begin by uploading the trained model and use it for exploration. Issue a negative reward from the controller when the vehicle deviates from the line, and issue a positive reward when the vehicle remains close to the line. If the vehicle deviates excessively, halt the exploration, reposition the robot onto the line, and start again. The controller can still compel the robot to approach the track while in autopilot. This phase demands considerable time and requires the undivided attention of the supervisor. Continue until enough data is gathered to update the model.

5. **Update the model:** Move the collected data to their corresponding repository, most should go into training. Launch the notebook for training the model. Upload the updated model to the OpenBot app. Note: To avoid losing all progress with a faulty new model, it is good practice to save the previous model, especially if it is showing promising result.

6. **Repeat:** Repeat step 4 and 5 until obtaining a satisfactory result or determining that the process is unsuccessful.

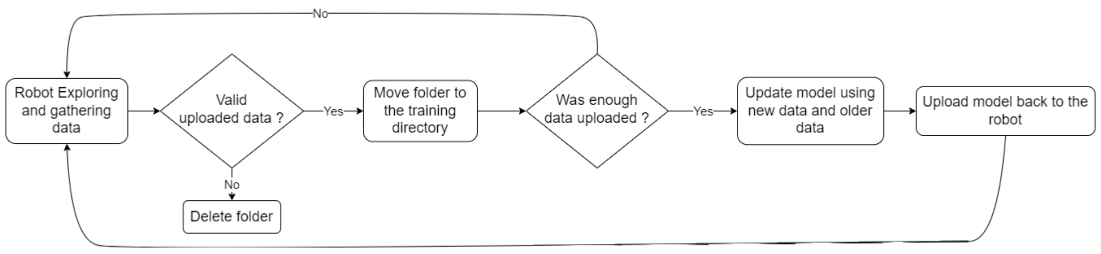

No
Robot Exploring and gathering data
Valid uploaded data ?
Yes
No
Delete folder
Move folder to the training directory
Was enough data uploaded ?
Yes
Update model using new data and older data
Upload model back to the robot

Figure 5.3: Simplified Flow Chart of the Reinforcement Learning protocol in the preliminary experiment.

Experiment with Supervised Machine Learning

1. **Setup:** Turn on the python server following the guide in the OpenBot github [28]. Make sure the phone and computer are connected to the same network and the OpenBot app connects to the server. Note: Verify that the phone is not in 'Lock Orientation' mode.

2. **Data collection:** Collect approximately 20 minutes of data of the robot making laps. For better results, take more at different times of day and weather.

3. **Train the model:** Move the collected data to their respective directory. Keep the best data for testing, and delete all folders with poor data. Train the model using 'pilot_net' as model.

4. **Upload the model:** Upload the new model to the phone. Use it with autopilot and see how it performs.

5.1.6 Results

In this subsection the results from the described method are given, as well as results on the same task using the Supervised Learning method, expected for OpenBot.

Results using Reinforcement Learning:

Training Performance

The experiment protocol was followed. Approximately 5 minutes of data was collected, 3 minutes for training and 2 minutes for testing. The first model, named 'reinforcement_learning', was trained and uploaded. The model ran and continuously collected data until it deviated significantly from the circuit, resulting in significant negative rewards (-15). Manual intervention was required to bring the robot back on track.

Every 2 minutes of data collected triggered an update of the model. It took approximately 6 to 7 updates before any noticeable progress was made in decision making. Due to the predominance of left or right turns in the laps, the model struggled to turn in the less common direction. Guiding the robot, imposing penalties for undesirable behaviour and selectively adding data to address incorrect decisions helped to mitigate this problem.

However, an unresolved challenge remained - the robot would continue straight ahead if it lost sight of the track, posing a safety risk and cost times with bad data collection.

Task Performance

The conclusive model demonstrated the ability to follow the line and complete up to 4 laps before losing track. The model stayed close to the line and appeared to decelerate during turns. Issues appeared particularly on the side of the circuit closer to the windows with bright lighting, where the likelihood of losing track and keep running straight was greater.

Time Efficiency

Achieving successful laps required about 30 minutes of data, but training from scratch took a total of 8 to 10 hours.

Sensitivity Analysis

As previously noted, the model showed difficulties in areas illuminated by natural white lighting, affecting the robot's behavior. Moreover, the robot exhibited susceptibility to specific changes; for instance, adjusting the position of a red handbag led to the robot missing a turn. However, the removal of a wooden box or a chair did not appear to have a discernible impact on the model.

Results using Supervised Learning

Training Performance Still following the protocol, a total of approximately 20 minutes of data was collected, focusing on keeping only high-quality data without deviations from the line. The model underwent training using the Python Server and was then uploaded to the phone. There were no notable challenges with this method, with the exception of occasional ground slipperiness causing long runs of data collection to become unusable.

Task Performance

The Supervised Learning model exhibited the capability to complete up to 7 laps before deviating too significantly from the line and encountering obstacles. Interestingly, the model did not consistently rely on the line as a guideline but rather seemed more attuned to details from the surrounding environment. While successfully navigating all turns, the model spent minimal time on the line itself. Notably, the performance of the model was highly impacted by various lighting conditions and weather.

Time Efficiency

The entire process of data gathering and model training took approximately 2 hours. Notably, the familiarity with the circuit significantly contributed to the success of data gathering time efficiency.

Sensitivity Analysis

As previously highlighted, the Supervised Learning model demonstrated sensitivity to various lighting sources. To mitigate this, data collection was conducted at different times of the day and on diverse weather conditions. The model also exhibited high susceptibility to moving objects, such as a cat passing by.

5.1.7 Conclusion and Discussion

Creating a reward function, adding it to the loss function of the model in order to maximize it resembles the RL algorithms but some important parts were missing. The model running on the circuit after a short prior training and kept being updated to perform better without being controlled by a human and learning from its experience made it appear like reinforcement learning. However, it was not using the typical RL framework and was not implemented as a MDP. In the end it is more similar to Supervised Learning with an additional reward since 'bad' data, meaning instances where the robot strayed from the line, was not used during learning but was instead deleted. In RL, the 'bad' data is as valuable as good data so it learns what not to do but since the update imitated Supervised Learning, bad data was usually removed. The model learned using training data and testing data to see if the model is able to make correct predictions in never seen before data. That is a process specific to supervised learning and not used in reinforcement learning. The experiment was not properly designed from the start to make a RL algorithm, there were no Markov Decision Process and well-defined actions or states. While RL is possible with continuous action and states space, there are usually techniques to surpass that such as discretizing the action space by using uniform distribution for different discrete intervals.

Despite the misconceptions, this preliminary experiment was a crucial part for this study. It was a first approach to coding in Java for Android Studio, designing a reward function and noticing its challenge, working with OpenBot and understanding all the already existing code. It allowed to see what are some of the challenges of implementing reinforcement learning to Openbot, such as the luminosity issue with the reward function, the bad design for a RL algorithm, the complexity of a continuous action space. Moreover, the hope with this project was to implement RL directly on the phone and this preliminary work allowed to start the new experiment with a better base, knowledge on what still needs learning, new objectives and hypothesis.

5.2 Implementing Reinforcement Learning Algorithms In Different Scenarios

This section discusses the implementation of different RL algorithms in two different scenarios and their relation with the implementation of the algorithm on OpenBot. The codes are available on the repository [31] under the $Cartpole$ and $Pong$ folders. In the Cartpole scenario, deep Q-learning is used to train the model and in Pong, the policy gradient algorithm is used. The results of the training for each agent is in a video showing the agents play one episode of their respective game [32].

5.2.1 Cartpole

Cartpole is a simple game with a pole placed on a cart that can only move on the horizontal axis as shown in Figure 5.4. The aim is to keep the pole attached at equilibrium vertically above the cart. To simulate Cartpole, the $gymnasium$ library from OpenAI [33] is used. The tutorial from Pytorch [34] was used for this study and will be rapidly presented. Only a few modifications had to be made for the model to work nicely. The algorithm followed for training this agent is Deep Q-Learning as described in the section 3.3.

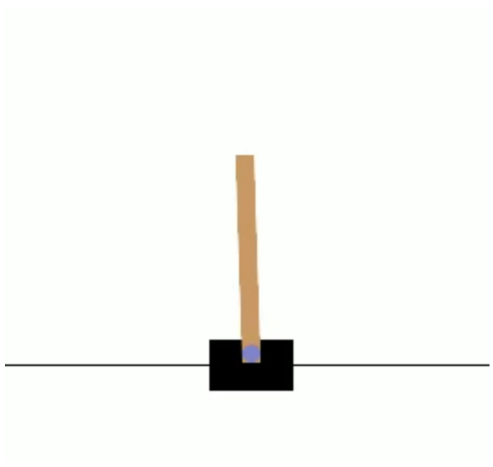


Figure 5.4: Screenshot of the Cartpole game with a cart attached on an horizontal axis and the pole vertically standing above.

Since Cartpole is a simple game, only a few parameters are provided for the state instead of the entire image seen in the figure. This greatly simplifies the computation since training a model to understand an image requires way more work than working on the 4 parameters making the state in Cartpole. The 4 parameters are:

- Cart Position $\in [-4.8, 4.8]$
- Cart Velocity $\in]-\infty, \infty[$
- Pole Angle $\in [-24°, 24°]$
- Pole Angular Velocity $\in]-\infty, \infty[$

The model is initialized as follows using the $torch$ libraries:

```python
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

It is a pretty simple model, with $n\_observations = 4$, 4 being the parameters discussed earlier and two possible actions $n\_actions = 2$; going right or going left. There are 2 hidden layers of size 128. The activation functions are both ReLU 3.4 and the optimization is AdamW also from *torch* libraries.

The loss $\mathcal{L}$ used is the *Huber loss* which is the mean squared error when the temporal difference error δ is small, and the mean absolute error when the error is large. The temporal difference error δ is the difference between the predicted value of a state or state-action pair and the actual observed value:

$$\delta = r + \gamma \max_a Q(s', a) - Q(s, a) \tag{5.5}$$

The *Huber loss* is then:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r)\in B} \mathcal{L}(\delta) \tag{5.6}$$

Where

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

B is a batch of transitions sampled from the replay memory. The replay memory stores the transitions that the agent observes and random selection of batches are used during training in order to decorrelate the transitions. The tutorial states that it stabilizes and improves the DQN training procedure.

The reward is computed by giving +1 for each time step until termination or reaching the threshold of 500 timesteps. Termination occurs when the pole angle exceeds its limit of either $-24°$ or $+24°$ or the cart position exceeds -4.8 or $+4.8$ on the x-axis.

Here is a simple MDP description of Cartpole:

- **States** ($\mathcal{S}$):
  - Cart Position $\in [-4.8, 4.8]$
  - Cart Velocity $\in]-\infty, \infty[$
  - Pole Angle $\in [-24°, 24°]$

  - Pole Angular Velocity $\in]-\infty, \infty[$

- **Actions** ($\mathcal{A}$):
  - Move left: 1
  - Move right: 2

- **Rewards** (R):
  - $R = +1$ at each time step without termination
  - $R = +0$ at termination

- **Termination**:
  - Angle exceeding limits
  - Position exceeding limits
  - 500 timesteps reached

Figure 5.5 is the diagram that illustrates the overall resulting data flow. In blue there are actually two models that are initialized identically, the Policy Net and the Target Net. The Policy Net is the model deciding the action and calculating $Q(s, a)$. The Target Net is a slightly 'older' version of the Policy net used to compute the expected value of the actions $r + \gamma \max_a Q(s', a)$ when computing the temporal difference error δ seen in Eq 5.5. The algorithm uses ϵ-greedy method for exploration.

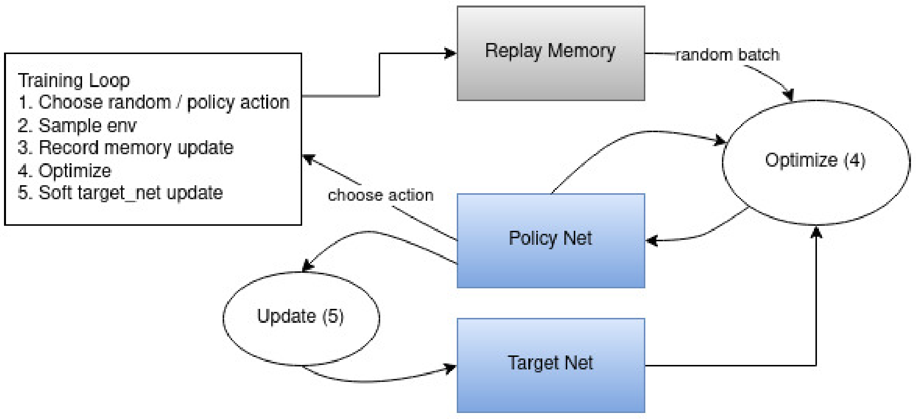


Figure 5.5: Data flow of the algorithm for training a DQN agent to play cartpole. Diagram taken from the tutorial [34].

With a GPU, the training can last between 5 and 10 minutes but results vary from one training to the other because of the randomness in the exploration. Most of the time, the model is able to converge and last for the entire 500 timesteps without termination in less than 600 episodes, but that is not always the case. Different results of training can be seen on the figure 5.6, the code is the same for all four examples.

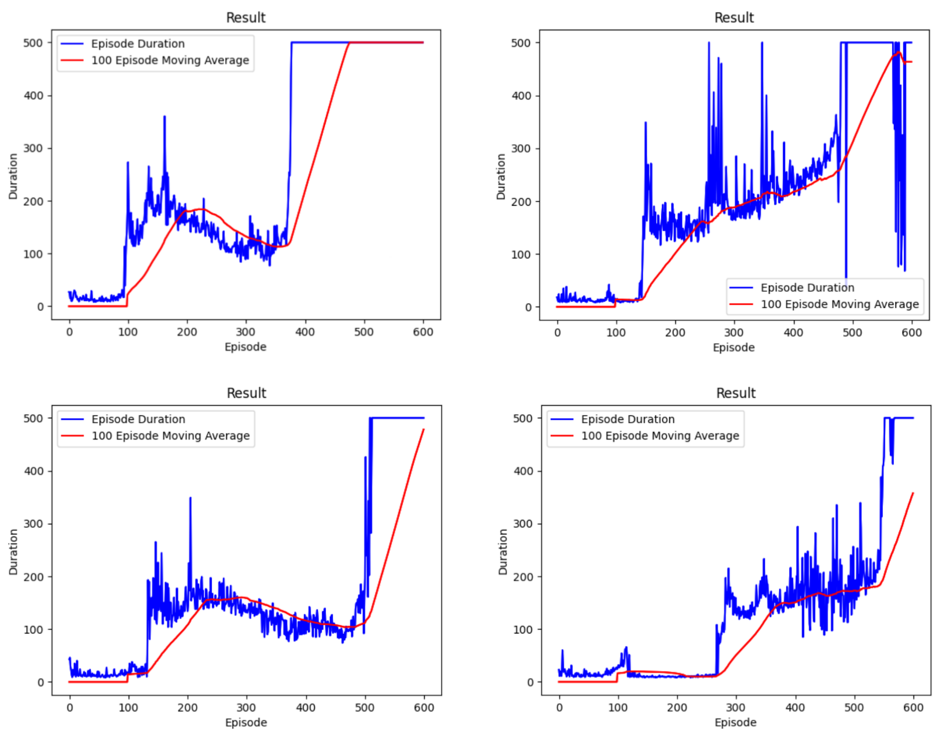


Figure 5.6: Graphs of 4 different instance of training the model on Cartpole, with the same hyperparameters for each, showing how training varies from one training to another.

5.2.2 Pong

Pong is a famous arcade game simulating a game of Ping-Pong where two opponents try to score by sending a ball in the opponent's goal. Each player can move on the y-axis and if the ball gets behind their respective paddle, the opponent scores a point. The first to reach 21 points wins the game. Unlike the Cartpole scenario, this time the agent will learn to play Pong with the pre-processed image of the game as its input. A rendered image of the agent playing the game can be seen in Figure 5.7.

The agent was trained using policy gradient introduced in the section 3.4. In this study, the tutorial from Omkar Vedpathak [35] was followed and the code from his repository [36] used. Omkar's work was itself heavily inspired on the work of Andrej Karpathy [37]. However, the code and tutorial are from 2019 and most of the code is depreciated and required an update to work. The original version from Omkar is presented, then the modifications made to adapt better to the need for this study are described. The two are compared in the end of the section. But first, the base from both codes is introduced.

Base for both algorithms

In Pong, the reward is simple; for each point scored by the agent, a reward of +1 is given and for each point scored against the agent the reward is -1. This means that if the final reward is negative, the agent lost the game and, similarly, if the reward is positive the agent won the game. The best possible score at the end of an episode is +21 meaning the agent score 21 to 0 and the worst is -21 meaning the agent was not able to score a single point. The agent is training against a computer programmed to follow the ball position on the y-axis but has a maximum velocity so that it can actually lose the game.

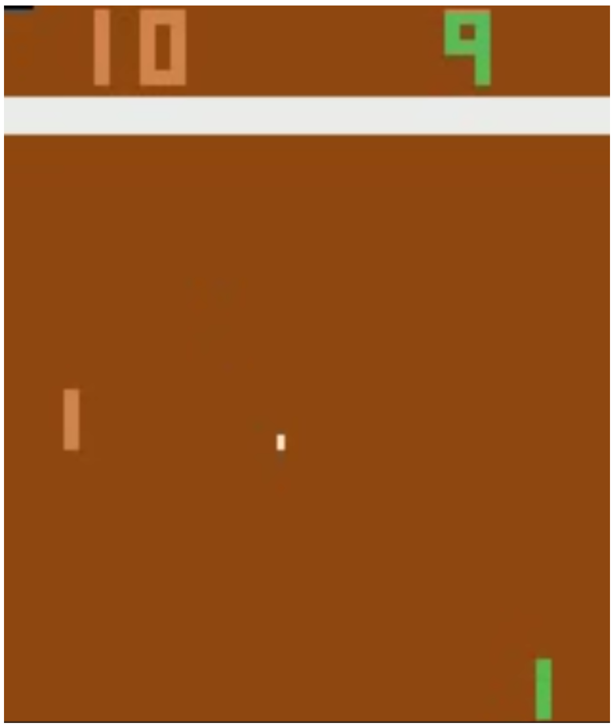


Figure 5.7: Picture of Pong-v0 from the OpenAI gymnasium library [33] rendered in rgb_array. On the left, in brown, is the computer simply following the ball and playing against the agent in green, on the right.

Here is a simple MDP description of Pong:

- **States** ($\mathcal{S}$):
  - Pixel representation of the game screen
- **Actions** ($\mathcal{A}$):
  - Move up: 2
  - Move down: 3
- **Rewards** (R):
  - $R = +1$ if agent scores a point
  - $R = -1$ if opponent scores a point
  - $R = 0$ for all other time steps
- **Termination**:
  - Opponent reaches 21 points
  - Agent reaches 21 points

During training, a discounted reward is used in order to attribute more accurately the reward with the action that is likely an important contributor to the reward. This is because the action just before losing a point are not as relevant if the paddle was far away from the ball anyway, the paddle should have moved sooner towards the ball and that is why discounting the reward depending on the timestep is relevant. Here is the equation for the discounted reward R_t so each action has its corresponding reward:

$$R_t = \sum_{k=0}^{T-t} \gamma^k \cdot r_{t+k} \tag{5.7}$$

With γ the discount factor.

Both models have 3 layers, the input layer, an hidden layer of 200 nodes and the output layer. ReLU (refer to Equation 3.4) is the activation function in the hidden layer in both cases but they use different activation function on the output layer.

They both use policy gradient to update, so they both use gradient ascent at the end of each episode/game to update the model. The image is also pre-processed in order to simplify the problem, only the relevant part of the image is kept, the background is erased and the image is collapsed into a simple column vector of size $75 * 80$ so of size 6000. Since it is important to know the movement of the ball and paddle, the image fed to the model is actually the older image subtracted to the new image so only the difference between the two remains, except for the first image since there are no previous image.

The Root Mean Square Propagation (RMSProp) is an optimization algorithm used when updating the models. RMSProp helps to scale down the learning rate for parameters that have large gradients and scale up the learning rate for parameters that have small gradients. It is often implemented as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{5.8}$$

With

- θ_t: The parameters (weights) at time step t.
- η: The learning rate.
- $E[g^2]_t$: The decaying average of the squared gradient at time t.
- ϵ: A small constant added to avoid dividing by 0.
- g_t: The gradient of the loss function with respect to the parameters at time step t.

However, since the aim is to maximize the return, the equation needs to be slightly changed to be using gradient ascent:

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{5.9}$$

Both methods use the same following hyperparameters:

- H = 200 : number of hidden layer neurons
- batch_size = 1 : used to perform a RMSprop param update every batch_size steps
- learning_rate = 1e-3 : learning rate used in RMS prop
- gamma = 0.99 : discount factor for reward

- decay_rate = 0.99 : decay factor for RMSProp

Finally, there a three possible actions when playing Pong, moving UP, moving DOWN or staying at the same place.

Original Method

In the original code from Omkar [36], the model is simplified with having only one output: the probability of moving UP. If this probability is high enough, the paddle goes UP, otherwise it goes DOWN. The model is described in Figure 5.8.

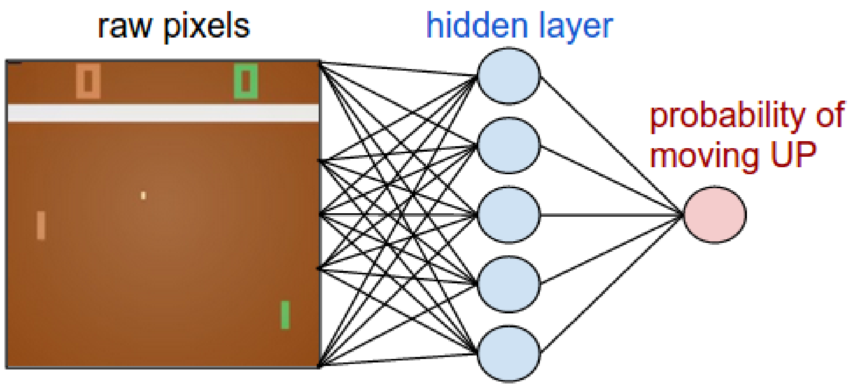


Figure 5.8: Illustration of the ANN model trained to play Pong. Image from Andrej Karpathy's blog [37].

In his code, Omkar does not use ϵ-greedy exploration, instead the action is decided as follows:

```
action = 2 if np.random.uniform() < aprob else 3
```

With 2 being going UP, 3 going DOWN and aprob being the probability of moving UP computed by the model. This means that instead of using a decaying ϵ, this algorithm 'forces' the model to give high values for the probability of moving UP in order to be certain the paddle will move UP and low value for it to go DOWN. However, there is still a random factor that can make the paddle go UP when the probability is low and vice-versa. Nonetheless, as the model learns, its value for the probability will become either very close to 1 or very close to 0, making the randomness less likely to occur.

The activation function used for the output is the sigmoid function (refer to Equation 3.2), if the value given to the activation function is negative, the probability will be less than 0.5 and if positive higher than 0.5. This works well with this singular output.

The weights are initialized using Xavier or Glorot [26] initialization described in the section 3.2 and given equation 3.6 as follows:

```
model = {}
model['W1'] = np.random.randn(H,D) / np.sqrt(D)
model['W2'] = np.random.randn(H) / np.sqrt(H)
```

In here, H corresponds to the number of nodes/neurons in the hidden layer and D the size of the input. The shape of $W1$ is $(H\text{x}D)$ so $(200\text{x}6000)$ and the shape of $W2$ is $(200,)$. While it can be managed during the forward pass, the shape of $W1$ is a bit counter-intuitive. If reading the model in Figure 5.8 from left to right, first, the image of size 6000 is fed to the model and goes through the weights $W1$. Then, from the hidden layer of size 200 and after the activation function, it is multiplied by the weights $W2$ to give the output. It would be more intuitive to have the weights $W1$ set as $(D\text{x}H)$ for clarity.

Nevertheless, this method works well with the agent being able to beat the computer most of the time after 8k episodes. For the sake of testing where it would converge, a model was trained on 50k episodes and its results are shown in Figure 5.9.

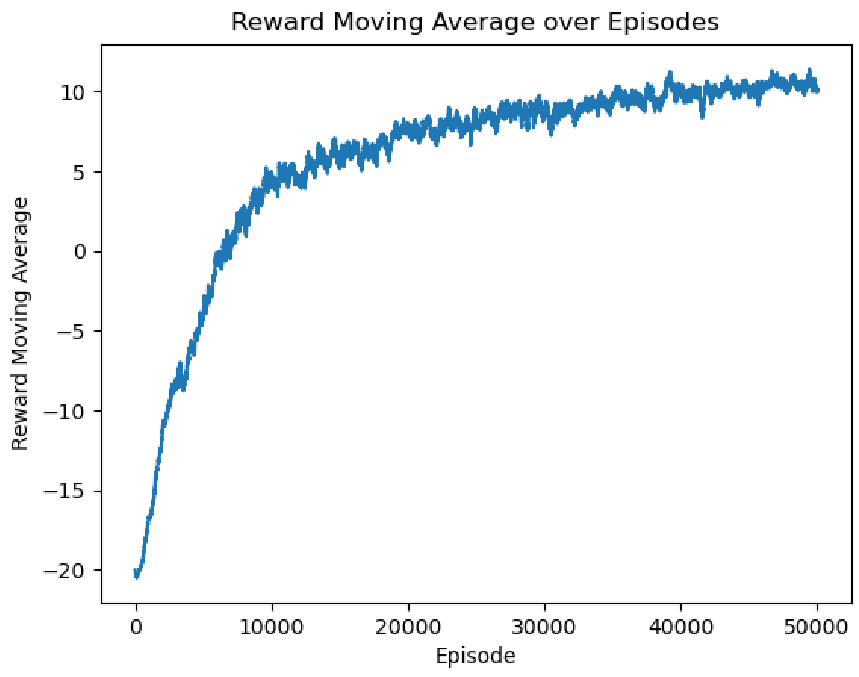


Figure 5.9: Plot showing the moving average of rewards over the last 100 episodes during the 50k-episode training.

The agent is able to beat the computer with an average of 21-11 at the end of training and appears to still be learning slowly. However, the training gradually takes more time as the episodes get longer with one round sometimes counting up to 10 exchanges. Training over 50k took around 50 hours to complete, but no GPU is set for this code since the model is created manually (forward and backward pass both manually implemented).

Modified Method

To make the algorithm more relevant to this study's experiment, the output of the model was changed from a singular probability of going up to two outputs: probability of going up and probability of going down. While the change seem unnecessary for this task, this was important to understand how to design the model for the experiment on OpenBot. Further explanation can be found in the following section 5.2.3.

To change the output to an array of probabilities with a size of 2, the softmax activation function was used (refer to Equation 3.5). Moreover, the method for exploration had to be changed and ϵ-greedy was implemented. The new action selection is done as follows:

```python
def choose_action(output, epsilon):
    is_random = False
    if np.random.uniform() < epsilon:
        is_random = True
        return np.random.choice([2, 3]), is_random
    else:
        action = np.random.choice(len(output), p=output)
        return action + 2, is_random
```

Here, there is a probability of ϵ that the taken action is random and a probability of $1 - \epsilon$ that the action will be selected depending on the probability of each action. The action selected is not always the one with the highest probability. Nevertheless, if the model assigns a probability close to 1 to one of the action, it is more likely to be selected than the other. The new boolean is_random is used when updating the model because when the action taken is random there's no explicit strategy or intention behind it and it needs to be reflected in the update. The outcome is still taken into account, but since the decision did not depend on the probability given by the model, the probability is not included in the computation for this action when updating the model.

For this model, ϵ was initialized at 0.15 with a decaying rate of 0.999. At the end of each episode, it is updated by multiplying itself with the decay rate.

Since the model has two outputs, the weights and layers in the model had to be modified. The issue of counterintuitivity raised before with the initialization of the weights was also taken care of, following is the new initialization, taking into account the modified output:

```python
    model = {}
    model['W1'] = np.random.randn(D, H) / np.sqrt(D)
    model['W2'] = np.random.randn(H, 2) / np.sqrt(H)
```

The size of the input is D, the shape of $W1$ is $(D\text{x}H)$, the shape of $W2$ is $(H\text{x}2)$ and the output is of size 2. The forward and backward pass had to be slightly adapted to this change but remain very similar to the original model.

The original model performed slightly better overall than the modified model when using the same hyperparameters. Both model were able to beat the computer, however, while not apparent on the Figure 5.10, the performance of the new model fluctuated more than the original model with sometimes winning 3 games in a row with +10 as final reward to losing a game with a -19 final reward. It is important to note that the model can vary like discussed in cartpole even with the same hyperparameters due to the randomness in the exploration and the variety of situations in the environment (Pong). Moreover, the hyperparameters might not be optimal, but training the model is very time consuming with up to 8 hours of training for 10k episodes.

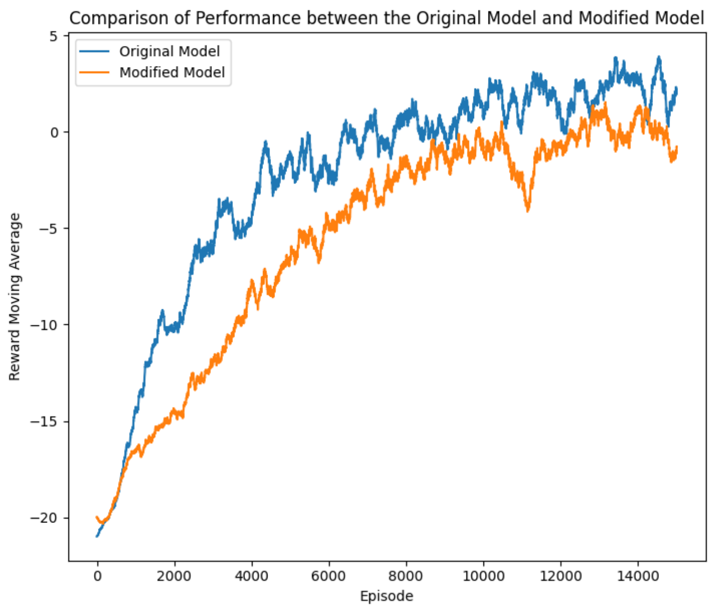


Figure 5.10: Graphical representation of the performance of the two models, in blue the original and in orange the modified model with two outputs. Performance is calculated using the reward moving average of the last 100 episodes and the models were trained on 15k episodes.

5.2.3 How These Examples Help Prepare for the Implementation of Reinforcement Learning on OpenBot

The implementation of these examples served two purposes: getting more familiar with reinforcement learning and its algorithms and using interesting features of each to the new experiment on OpenBot. Before starting this research, there was no prior knowledge in reinforcement learning beyond a very basic explanation provided during a machine learning course. This is the reason why the preliminary approach was incorrect and required an entire rework. Implementing the algorithms on simpler (Cartpole) or more studied (Pong) environments greatly helped with the understanding of what was lacking in the prior experimentation.

The interesting similarity with cartpole and implementing RL on OpenBot to remain on a circuit is that both aim to have the agent 'survive' as long as possible before reaching termination. The shaping of the reward with having a positive reward for each timestep the pole stays within boundary stated in the environment will inspire the reward shaping for the OpenBot experiment as it seems the most appropriate. However, the model and Deep Q-Learning is not feasible directly on the phone due to the lack of machine learning libraries and computing power required. Moreover, the input of the model is not a picture so the cartpole algorithm might not work with images as inputs. This is where the Pong code is really interesting: a working, simple and manually implemented model that can be imitated in Java on Android Studio. Moreover, the Pong scenario uses preprocessed images as input similarly to OpenBot. The algorithm used in Pong seems to fit pretty well to the new objective stated below and not too complicated to implement. The modified version of Pong is necessary since OpenBot has 3 different possible actions and not two; going left, going right or going forward. Testing it on Python before testing it on Android Studio and on the phone helps greatly with the debugging and finding clues on where difficulties may arise.

6 Methodology to Implement Reinforcement Learning on OpenBot

This Chapter describes how the reinforcement learning algorithm was implemented on the OpenBot app. The original code is available on OpenBot's GitHub [28], the code used in this study forked their code around September 2023 and, in order to avoid conflicts, was not updated with the official's code updates. The code for this study is available in a remote repository [31].

The code implementation of RL in OpenBot is done in the feature $PolicyGradientFragment$, the Model is a runnable class called $MyModel$ and used by $PolicyGradientFragment$. Finally the layout is a copy of the $LoggerFragment$ layout with some modifications to it.

6.1 Updated Experiment and Objectives

Learning from the preliminary experiment, an updated experiment with a new open circuit, method and objective was required. The mistakes from earlier are now taken into account for this update and a new base is designed.

It is important to view the new experiment as a RL problem, each run on the circuit is now called an episode and each time the model receives an image and selects an action is a timestep. To simplify the continuous action space, the possible actions $\mathcal{A}$ are reduced to 3 simple actions: moving right, left and forward. The robot is forced to move forward so it cannot gain rewards in remaining still and just spinning around will not let it remain on the track. The input is the pre-processed image seen by the smartphone mounted on OpenBot. In order to counter the issue with luminosity from the preliminary experiment, the circuit was modified to black plastic bags on the ground over a clear ground and white guidelines as seen in Figure 6.1. This serves two purposes, it counters the luminosity issue pretty well and allow a supplementary security feature of terminating the episode when leaving the circuit instead of manually terminating it. The entire circuit can be seen in the video showing the training of the agent [38].

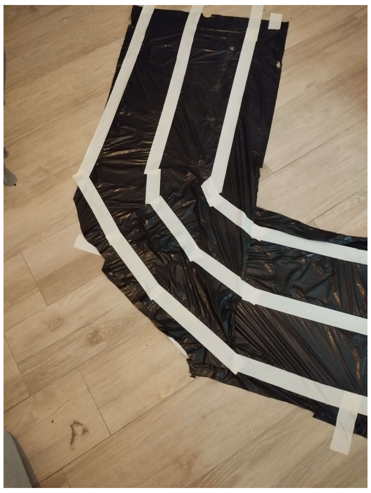


Figure 6.1: Picture of an extremity of the open circuit, displaying the setup of a black plastic bag over a clear floor and white tape to structure the track.

For the updated experiment, a simple Markov Decision Process to describe the RL environment is important to set the base:

- **States** ($\mathcal{S}$):
  - Pixel representation of what the phone mounted on OpenBot captures
- **Actions** ($\mathcal{A}$):
  - Move left: 0
  - Move right: 1
  - Move forward: 2
- **Rewards** (R):
  - $R = +1$ every 2 seconds remaining on the track
  - $R = +10$ if running for 15 seconds without termination
  - $R = -4$ if episode terminated before 15 seconds threshold
- **Termination**:
  - Exiting the circuit
  - Manual termination (emergency stop)
  - Running for 15 seconds

The new objective is to create a new feature on the OpenBot app for the entire RL process. Not using the pyhton server or python scripts. An ANN model needs to be manually implemented with the forward and backward pass. For this task, the policy gradient algorithm introduced in the section 3.4 and with Pong in the subsection 5.2.2 are used. A new reward function is created and termination when exiting the circuit is added. The aim is to see if the agent will be able to make turns and remain on the track for 15 seconds. Figure 6.2 shows the UI of the implemented new feature for this experiment. The new feature was implemented using the method presented in subsection 5.1.3.

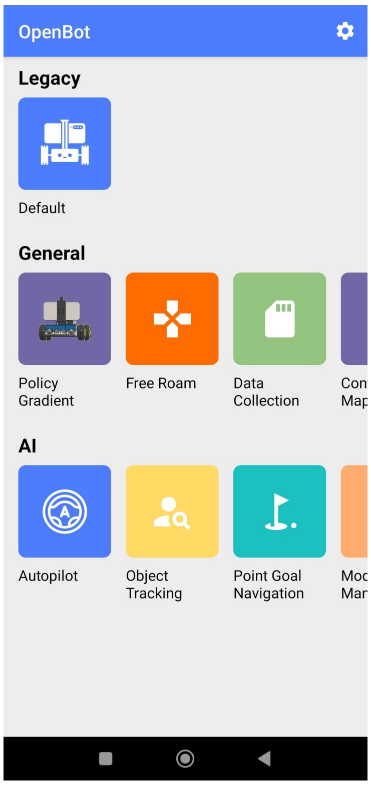


Figure 6.2: Screenshot of the new policy gradient feature added when first opening the OpenBot app.

6.2 Implementing Reinforcement Learning Features

This sections introduces the implementation to OpenBot of the main components in reinforcement learning: the state, the reward, the actions, the termination and the exploitation-exploration strategy.

6.2.1 State

As introduced in section 6.1, the current state of the agent is the processed image from the camera on the smartphone. The image is processed to make the model and computation simpler, it only keeps the bottom of the image so it is not disturbed by objects in the room and only sees the ground close to it. To simplify it even further the image is binary, with 0 meaning white and 1 for black pixels. An example of the process image can be seen in Figure 6.3.

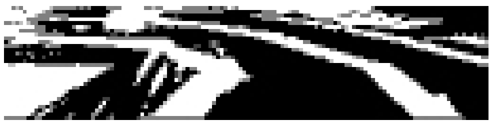


Figure 6.3: Processed image received by the model, the image is of size 128x30.

While the dark flooring was added to help with the luminosity issue, it is not matte and can be reflective in sunny days. Nevertheless, the threshold for black and white can be changed pretty easily by modifying the values 120 and 255 in the function $OpenCV Processing$:

```
    Imgproc.threshold(inputMat, inputMat, 120, 255,
     ↪  Imgproc.THRESH_BINARY);
```

If the training is done in the evenings instead of during the day, it is also a possibility to have the the real ground to also be seen as black pixels and then the only white pixels are the lines of the track seen in figure 6.1. The safety feature terminating when exiting the path still works using percentage of black pixels instead of percentage of white pixel threshold. This is done by simply replacing the return of $calculateWhitePixelPercentage$ by $100 - percentage$. In the code available in the remote repository, the image is cropped and resized to 128x30 in the $processFrame$ function and is turned into binary in the $OpenCVProcessing$ function.

6.2.2 Reward

The aim is to have the agent remain on the open circuit for 15 seconds. This objective is similar to the cartpole scenario where the aim is to keep the pole within boundary for 500 timesteps. At first, the OpenBot scenario was following the same rewarding process with +1 for each timestep still on the circuit. However, this caused exploding gradient, discussed in the subsection 6.4.5 and the reward was slightly modified. Instead of having +1 after each timestep, the reward is given every 2 seconds, so after approximately 20 timesteps. A negative reward is given upon termination by straying from the path or emergency stop of -4 and positive reward when reaching 15 seconds without straying of +10. The maximum total reward of an episode can reach 16, not 17 because the +1 at the 14th second turns into +10 for reaching termination. There is also the possibility to add +10 or -10 manually using the controller but it is usually avoided.

Similarly to Pong, the discounted reward is used when updating the model since the error of exiting the track is often due to a decision happening in earlier timesteps than the last one. However, timesteps from the beginning of the episode may not be responsible for exiting the track later on.

6.2.3 Actions and Termination

There are only three actions that OpenBot can select: going left, going right or going forward. This is implemented with a simple function $MoveAction$, if action is set to 0, the agent is turning left, if the action is set to 1, the agent is turning right and finally, if the action is set to 2, the robot is going straight. The function is simple, and sets the control to be sent to the robot, here is how going straight is programmed:

```
if (action == 2) {
                vehicle.setControl(0.45f, 0.45f);
                handleDriveCommand();
            }
```

Concerning termination, there are three possible termination:

- Exiting the track, so if more than 80% of the image is white when it is during the day or when more than 95% of the image is black during the evening. The difference in percentage is because most of the track is also black, so the threshold needs to be bigger to avoid stopping the agent while still on the course. This termination will give a negative reward, stop the agent and update the model.

- Staying on the circuit for more than 15 seconds, or 15000 millis in the code, will give a positive reward, stop the agent and update the model.

```
        if(elapsedTime >= LOGGING_DURATION_MILLIS )
```

- In case the first termination fails or something happens that requires to stop the agent, pressing (X) on the PS4 controller will stop the agent, give a negative reward and update the model.

6.2.4 Epsilon-greedy

For this implementation, while still using Epsilon-greedy, it is manually set by the supervisor and can be changed to be larger or smaller between every episodes. An entry is added to the layout to enter a number. If it feels like the agent needs more exploring then the supervisor can give a larger epsilon value. The model can also be tested in one episode by setting the epsilon value to 0. This method is implemented in the *ChooseAction* function. Figure 6.4 shows the UI of the implemented feature, the place to enter the epsilon value is visible as well as the different toggles available such as the 'Resume training' discussed later.

The *ChooseAction* function takes as input the output of the Model which is the probability for each action. First, a random value between 0 and 1 using the Random library and $random.nextDouble()$ is compared to the epsilon provided by the user. If the value is smaller than epsilon, the action chosen is random, otherwise, the action with the highest probability is selected.

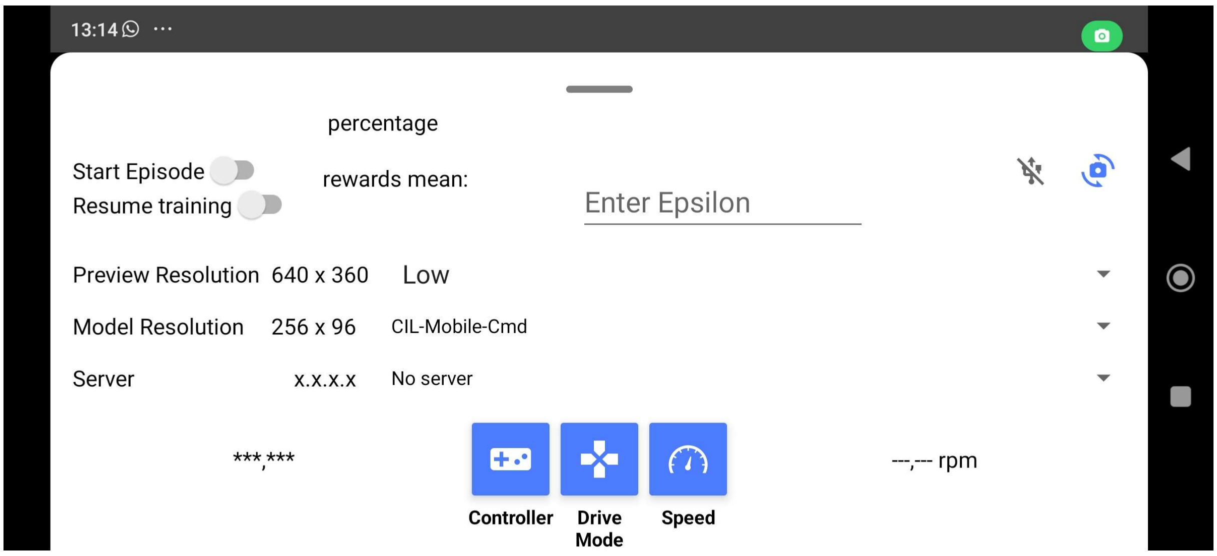

13:14
percentage
Start Episode
Resume training
rewards mean:
Enter Epsilon
Preview Resolution 640 x 360 Low
Model Resolution 256 x 96 CIL-Mobile-Cmd
Server x.x.x.x No server
,
---,--- rpm

Figure 6.4: Screenshot of the UI on a Redmi Note 11 of the POLICYGRADIENT feature showing the different possible interactions such as the possibility to enter an epsilon value and to toggle the resume training.

6.3 Creating the Model and Implementing Policy Gradient

This section discusses the initialization of the model, its weights, the forward and backward pass and finally, the implementation of the policy gradient for updating, using RMSProp.

6.3.1 Model initialization

The ANN model and its function are created in a separate class called $MyModel$. It can be initialized by giving two inputs: the dimension of the model's input and the dimension of the hidden layer. The input dimension is the processed image of size $128 * 30 = 3840$ and the number of nodes in the hidden layer is 200. The output layer is always of size 3 for the probability of each action. The weights are initialized the same way than for Pong, using Xavier/Glorot initialization [26]. The function is public so it can be used in the $PolicyGradientFragment$ and is called $initializeWeights$.

A new model is initialized if $Resume\ Training$ is not toggled. If it is toggled, the already existing model is taken instead and the model is not initialized again. After every episode, the model is updated and saved on the phone, replacing the older one. This is done using the functions $saveModel$ and $loadModel$.

6.3.2 Model Forward and Backward pass

The forward pass $policyForward$ and backward pass $policyBackward$ are the same as in the Pong example, however the data types work differently in Python than in Java. The forward pass takes the image as input and outputs the probability for each action. The Backward pass takes the states, hidden state, and the probabilities multiplied by the discounted reward and outputs the gradient.

It is discussed further in the challenges encountered implementing RL, section6.4, that using ArrayLists can be messy with Java since Numpy is not available and no suitable libraries for facilitating calculations and ArrayLists manipulation have been identified. So all the values are turned into Matrices for computation and then turned back into $double[][]$ for the weights of the model. The ReLU 3.4 and softmax 3.5 activation functions are also manually implemented as private functions of the $MyModel$ class. The functions are not optimized and most of the time iterate through each values of the arrays, fortunately the Java programming language is quite fast with computations.

The specific codes for the Forward pass and Backward pass can be found in the appendices 10.3.

6.3.3 Gradient Ascent

The Gradient Ascent is the same as in Pong, following the equation:

$$\theta_{t+1} = \theta_t + \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{6.1}$$

With

- θ_t: The parameters (weights) at time step t. (modelArray in the code)
- η: The learning rate.

- $E[g^2]_t$: The decaying average of the squared gradient at time t. (the RMSPropCacheArray in the code)
- ϵ: A small constant added to avoid dividing by 0. ($1e-5$ in the code)
- g_t: The gradient of the loss function with respect to the parameters at time step t. (g in the code)

Here is the code used to update the model using Gradient Ascent, it is also shown to demonstrate how the functions aren't optimized with iteration through each values of the array, the update can take 2 to 3 seconds.

```java
public Map<String, double[][]> updateModel(Map<String, double[][]>
 ↪  gradBuffer, Map<String, double[][]> rmsPropCache, double
 ↪  decayRate, double learningRate) {
        Map<String, double[][]> updatedModel = new HashMap<>();
        for (String key : model.keySet()) {
            double[][] g = gradBuffer.get(key); // Gradient

            // Update rmsProp_cache
            double[][] rmsPropCacheArray = rmsPropCache.get(key);
            for (int i = 0; i < g.length; i++) {
                for (int j = 0; j < g[0].length; j++) {
                    rmsPropCacheArray[i][j] = decayRate *
                     ↪  rmsPropCacheArray[i][j] + (1 - decayRate) *
                     ↪  Math.pow(g[i][j], 2);
                }
            }

            // Update model
            double[][] modelArray = model.get(key);
            for (int i = 0; i < g.length; i++) {
                for (int j = 0; j < g[0].length; j++) {
                    modelArray[i][j] += learningRate * g[i][j] /
                     ↪  (Math.sqrt(rmsPropCacheArray[i][j]) + 1e-5);
                }
            }

            // Reset batch gradient buffer
            gradBuffer.put(key, new
             ↪  double[modelArray.length][modelArray[0].length]);

            // Store updated model
            updatedModel.put(key, modelArray);
        }
        return updatedModel;
    }
```

6.3.4 Training Process

The training is simple and requires the supervisor to be positioned where they can see the entirety of the open circuit. The learning process is summarized, simplified and explained in the flow-chart seen in figure 6.5. The figure represents what happens during one episode of training, it is repeated until the agent is able to remain on the track for more than 15 seconds or if no changes are noticed after numerous episodes. The exploration vs exploration is ϵ-greedy in this scenario. Termination is either exiting the track, lasting more than 15 seconds or emergency stopping.

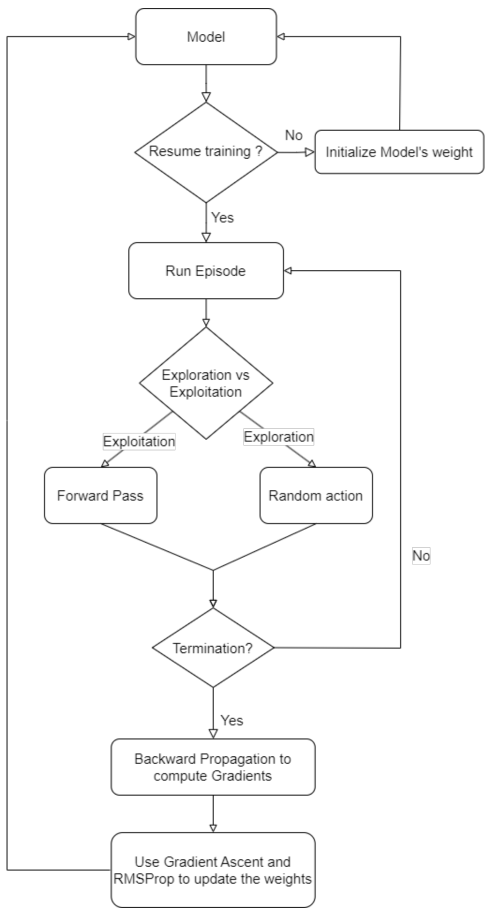

Model
Resume training ?
No
Initialize Model's weight
Yes
Run Episode
Exploration vs Exploitation
Exploitation
Exploration
Forward Pass
Random action
No
Termination?
Yes
Backward Propagation to compute Gradients
Use Gradient Ascent and RMSProp to update the weights

Figure 6.5: Simplified Flow Chart of an episode in this experiment.

6.4 Difficulties Arising with Implementation

This section presents some of the difficulties encountered while implementing RL on the smartphone and what can be expected when trying to replicate the experiment.

6.4.1 Using ArrayLists

While it is intuitive to work with Python arrays and lists, it is a bit more complex in Java for those used to the Python programming style. In Java, a versatile datatype is the ArrayList, capable of holding various types of data such as double, int, double[], double[][], RealMatrix and more. However, manipulating values inside of the lists is more complicated. It often requires to convert the ArrayList of double for instance to a double[] or an ArrayList of double[] to double[][]. It requires multiple functions to manipulate them, the rewards is an ArrayList of double, the hidden state has to be stored in a vertical array, so in a double[][], same for the input x and the probability from the Model is a double[]. Each require a function to turn them back into arrays so they can be manipulated and another to put the new arrays back into the List. Moreover, as stated previously, most operation done with arrays like double[] require a function iterating through each values of the array. This meant more than 9 functions to work with the ArrayLists and it becomes rapidly confusing.

After looking for a simpler and more readable way of handling the values, the RealMatrix and MatrixUtils packages appeared as the simplest solution. The packages can handle vertical or horizontal matrices and make the basic calculations such as multiplication and transpose without requiring a manually implemented function. It makes the code more readable, ArrayLists are still used since appending the matrix after each step only requires a simple $.add()$ to it. The packages also can turn arrays into matrices and matrices back to arrays in a simple line.

6.4.2 Errors with the Random Action

After a few trials of training, the model always appeared to favor turning left. And this even when the first turn was going right. After further inspection, it appeared the random selection of action was not set correctly and would only steer the vehicle to the right and forward. It seemed the model was countering the random actions by turning left.

6.4.3 Crashing if Restarting too Fast

Another issue when training is restarting the training before the model is updated. The model is quick to update, at most it can take a few seconds. However, the button to the emergency stop is the same as the starting button, if it is pressed in fear of the exiting circuit termination failing and the termination is not failing, it will crash the app. The app crashing is not too bad, but the epsilon value needs to be set again and if the $Resume\ Training$ is not toggled, the previous model can be replaced. It requires to be careful but it is manageable, although can be time consuming.

6.4.4 Time and Battery Consumption

Each training episode lasts less than 15 seconds, and resetting the robot takes only a few seconds using the controller to reposition it. If the robot requires as many episodes to complete the route from different starting points as it does to beat the opponent in Pong, then around 8,000 episodes would be required. With 15 seconds total for the episode and repositioning the robot it would take around 33 hours. Unlike Pong, this training needs to be supervised and Pong's training is already verified to work properly. Additionally, the smartphone needs to charge after around 4 hours of training, OpenBot's battery lasts around 20 hours but takes longer to charge. In that timeframe, the weather or luminosity change. However, if the model is capable of learning, it should be able to adapt to new environments, although this requires additional training and always exploring. This means a simpler task than Pong is required with lower expectations in order to have a shorter training time.

6.4.5 Exploding Gradient

Another issue is exploding gradient, which occurs when the gradients in a neural network become so large during training that they cause instability and hinder the learning process. While gradient explosion did not occur when training Pong, it occured when training OpenBot. A possible explanation is that the rewards are given too frequently or are too large. Since Cartpole is not updated the same way as Pong, spacing the reward to every second or two instead of every timesteps helps with gradient explosion. Finally, reducing the learning rate also helps with exploding gradient. So instead of using 10^{-3} as the learning rate, 10^{-5} is used during the training of this thesis. However, even with these modifications, the exploding gradient still inexplicably occurs. It is maybe happening when the data is corrupted by a crash or instant termination.

6.5 Protocol

This section describes the protocol to repeat the experiment from this thesis:

1. Start training by placing the robot at any extremity of the track.
2. Connect a controller to the smartphone using Bluetooth.
3. Turn on the OpenBot app and select the PolicyGradient icon.
4. Choose an input for epsilon, in this thesis the initial value for epsilon is 0.5.
5. Start a first episode, pressing (X) on the PS4 controller to start the episode.
6. Pressing (X) while the episode is running to terminate the episode in case the safety for exiting the circuit fails to terminate.
7. Check the processed images in the OpenBot file in the smartphone to see if the threshold for the binary image is set correctly for the current luminosity.
8. If necessary, modify the threshold as seen in the subsection 6.2.1.
9. When this is done, move the robot back to the extremity of the track, either by hand or using the controller.
10. Start the first episode again until termination.
11. Toggle ”Resume Training” to train the model, otherwise a new model with new weight is initialized every time and the updated model is replaced.
12. After each termination, put the robot back to the starting place, without being too precise on the starting position and orientation.
13. Repeat episodes until the agent succeeds the first turns multiple times without exiting the track.
14. Decrease the epsilon value accordingly.
15. To check how the model is doing so far, set the epsilon to 0.
16. In order to avoid losing a good progress of the model, save the model on the smartphone in a new folder inside the OpenBot folder every 100 episodes or so.
17. Repeat episodes and modifying epsilon accordingly until the agent is able to remain on track or until no further progress is observed.

7 Ensuring Safety of the Robot and its Surrounding

This chapter presents the different safety measures taken to avoid damaging the robot and its environment without impacting the training. This is a very important part in training a RL model in real-world environment since random action can cause the robot to crash, fall down the stairs or hit someone.

7.0.1 Manual Emergency Stop Using Controller

The first security measure implemented was an 'emergency' stop on the controller. By pressing the (X) button on a PS4 controller when the model is running and the robot is moving will immediately send the order to the motors to stop. It is a pretty simple code but important to have with any robot/machine active in real-world environments. This feature does not require to be very close to the robot and able to reach it, it can be done at a reasonable distance as long as the Bluetooth connection is still activate.

7.0.2 Automatic Stop When Exiting the Open Circuit

Another safety measure implemented for the new Experiment is to terminate the episode if the robot exits the circuit. As seen in the Figure 6.1, the flooring is quite clear with a dark plastic representing the path to follow. The image is processed and is then in black and white. The percentage of white pixels is computed and if it exceeds 80% the motors are ordered to stop and the episode is terminated. This appear to be working quite well if there is enough lighting on the circuit but is not foolproof, if the OpenBot strays too far without terminating the episode, the manual emergency stop should be used.

7.0.3 Pre-existing feature on OpenBot

OpenBot has 2 push buttons on the front and the back of the vehicle for obstacle detection. When either of the 4 detector is pressed, the motor stops and some LED lights up to alert the vehicle is in contact with an obstacle. While this prevents the robot to forcefully move against an obstacle, the object often has to be hit before the robot stops and could damage the robot or its push button. In a previous project, one of the push button was damaged and started to stop the motors just with the friction of moving forward. Fortunately, the button is easily replaceable and cheap.

8 Results

This chapter presents the result of training an agent to remain and move across a simple track.

8.1 Training Duration and Variability

The training takes around an afternoon but the total time depends on the initialization that has a random component to it. Sometimes the model initializes quite nicely and other times it will be more likely to select one action than another. The training time is also dependant on the random actions since it can take a long time before random actions help make the first turn. In general, the training takes 3 to 4 hours.

8.2 Learning Progress and Performance Milestones

The experiment was conducted 15 times and the model training was successful 8 times. The other 7, either a crash of the app caused a bad update of the model which is hard to recover, or exploding gradient made it impossible for the model to continue learning with new data.

In a successful training, it takes 50 to 100 episodes for the model to start learning to do the first turn, this is dependant on the random actions and initial weights. After 100 more episodes, the agent rarely misses the first turn but often has a tendency to prefer turning in the same direction even when the turn is completed. After approximately 300 episodes, the agent is starting to remain on the track more than 15 seconds although rarely. This can also happen randomly before thanks to the random actions, however, it is very unlikely to repeat. Finally, after around 400 episodes, the agent is able to remain on track for 15 seconds half of the time, sometimes missing a turn that seemed like it was able to do easily before just from a slight shift to the right or left. Training the agent with more episodes will often deteriorate its performance. When the agent starts to miss the turns more frequently than before, it is time to stop the training since it will rarely go back to its peak performance.

These results and the progress during training are summarized in a short video [38].

8.3 Crashing and Losing Models

The mobile app sometimes crashes, either because the emergency stop button was pressed almost simultaneously with the termination from exiting the circuit, or because the robot was moved or a new episode started before the model finished updating. It can also occasionally crash when an episode is too short (immediate termination) because of empty arrays, even with a few prevention such as not updating the model if there are less than 5 timesteps. This can also cause to update the weights of the model with values set to infinity and the new output of the model become NaN (Not a Number). It is important to save the model in a folder inside the OpenBot folder when a milestone seems to be reached or at least every 100 episodes. If the model is saved elsewhere than the OpenBot folder, it will not be readable because there are no authorizations set up on the OpenBot app to read external storage.

9 Discussion

The primary aim of this thesis was to assess the feasibility of implementing reinforcement learning entirely on an Android smartphone. This meant verifying if the computation time will not hinder the ability of the agent to fulfill its goal and the overall time necessary for the implementation and training is reasonable. The secondary objective was to determine if this implementation could serve as a practical reinforcement learning project for students or enthusiasts. In this chapter, the results are discussed and interpreted, followed by an examination of the study's limitation. Then, some recommendations for future research or how to ameliorate this experiment are discussed. Finally, some personal notes about the choice for the algorithm and the results from the foundational experiments are discussed.

9.1 Interpretation of the Results

The results show that implementing RL on a smartphone is reasonably feasible. The training does not need to be conducted consecutively and can be divided into multiple sessions, with a total duration of 5 hours at most. Using a simple model updating after every episode is not causing any visible latency issue and the training is not unreasonably long. This is encouraging for implementing smartphones into robotics and the eventuality to create a practical RL project accessible to students and enthusiasts. OpenBot being an affordable platform, it opens door for a project done showing reinforcement learning in a real world scenario and that could be tuned for different objectives than staying on a track such as obstacle avoidance or mapping.

Moreover, this study shows that modern smartphones can be used to train machine learning models without requiring prior knowledge of Java programming for mobile apps as long as one is willing to learn about the language. It also displays an interesting potential if linked with the work to implement the Robot Operating System (ROS) for Android [3]. ROS is highly popular in robotics and is now becoming accessible on smartphones which allows even more implementation of smartphones in robotics. Additionally, robotics is often taught using ROS, presenting another opportunity to create a practical project that integrates both reinforcement learning and robotics.

Finally, the model deteriorating after reaching a peak performance is not unusual in machine learning and ANNS. It can be caused by overfitting, where the agent will perform very well in a know route on the circuit but a small shift will make it perform poorly. The overfitting can be caused by the lack of regularization or excessive training without enough new scenarios. It is also possible that the model is too simple, without enough neurons to capture the features of the environment to allow a better performance.

9.2 Limitations of the Study

Although the study shows promising results in implementing reinforcement learning on a mobile app, there are some limitations to consider. First, the experiment was mostly conducted with one smartphone. The Feature was also tested on a Redmi Note 11 and appeared to work just as well. Nevertheless, all the complete training were done on the smartphone provided by the University: a Samsung Galaxy S22. It is important to note that the study might not be reproducible on some older phones with an Android version older than 11. A low-performance smartphone might also have a more difficult time with the computations.

Secondly, this experiment was done in only a simple scenario where binary colors were used and the robot is not moving at maximum speed, giving it more time to correct its trajectory. The algorithm might not work in other scenarios with more complex inputs.

Finally, there were still issues with the training such as exploding gradients and the application crashing. At its most successful state, the algorithm had only 8 out of the 15 trained model that were successful and this is without taking into account all the previous crash and problems during the implementation. Once the model has a problematic update with exploding gradients or due to a crash, it is usually irrecoverable and requires to either take an older version of the model if saved and hope exploding gradients will not occur or start over.

9.3 Recommendations for Future Research or Work

Future research could focus on finding the origin of the exploding gradients and overfitting to find ways to mitigate it such as using gradient clipping or L2 regularization. It also seems that the model could be more complex, provided that the code is optimized. Additionally, other algorithms or a different method than RMSProp to update could be tested to see if a faster convergence or better performance could be reached.

Other work could focus on making it an accessible and interactive feature to learn about reinforcement learning. Adding a simple interface allowing to change the size of the model, the hyperparameters, preprocessing of the image or the possibility to design a reward function would help users better understand reinforcement learning by letting them experiment with different aspects of the model. This approach could make learning more engaging and allow users to see firsthand how these factors influence the model's behavior.

9.4 Additional Notes

For this research, policy gradient was chosen for the reinforcement learning algorithm although deep Q-Learning was proven a powerful algorithm with the DeepMind's team demonstration on Atari games [17]. This is not due to the model being more complex in DQL, since as it was shown in Cartpole in the section 5.2.1 that only two hidden layers can be sufficient but because an update is done at each timestep. This is an efficient method but using a memory buffer and replay to update at each timestep would almost certainly cause latency when running it in the real world scenario. Indeed, in simulation, having a slow computation would only slow the rendering but the environment should not change faster than the agent in the simulation, so the agent will be slow but the environment will also slow down. This is not possible when training in the real world since the environment is not linked to the agent. Moreover, the agent would be still sending controls to the motors when it is computing what the next move should do and the environment would already be different when the new command is sent, this poses a safety concern. This is why the first implementation was a simple policy gradient model with only one hidden layer and 200 nodes. It can also explain why the model is not that performant either, with not being able to successfully remain on the track at every try but at most half of the time.

Finally, even though the aim of the study was to implement RL on a smartphone, a lot of work was put into the foundational experiments in chapter 5 that were not conducted directly on the smartphone. The three experiments were important for the final implementation but also had potential on their own. Although the preliminary attempt on OpenBot was not following the usual framework for reinforcement learning, the OpenBot app with the python scripts and server offer a great opportunity to implement a powerful reinforcement learning algorithm. With python and the GPU from the computer, the model used can be as complex as necessary and there are less concerns with the memories and computation power needed for training. The updating can be done sampling over all the data each time instead of only using the last episode and could converge faster.

The experiment with Pong from the section 5.2.2 was also very interesting to learn about reinforcement learning, machine learning and neural networks in general. Most codes available online are from a few years ago and are depreciated, requiring to change parts of the code. However, the code was malleable, easy to understand and work with. Every part of the code is accessible and not using libraries such as pytorch or keras as seen in Cartpole, makes it more readable and clear for a future user. Moreover, this was the only code found to successfully train the agent to play Pong after updating it, even after testing 8 different codes from various blogs and GitHub repositories. For any course or project on reinforcement learning, starting with this Pong algorithm is highly recommended. It has examples with multiple reinforcement learning algorithms and demonstrates how a successful RL training looks like.

10 Conclusion

In Conclusion, this thesis has shown the feasibility of implementing reinforcement learning on an Android smartphone. The results indicate that training RL models on smartphones are manageable, with reasonable computation and training time. However, the experience was conducted only on one smartphone although also tested on another. The reproducibility of the experiment is not guaranteed for older versions of Android or with a smartphone with lower computing power than a Samsung Galaxy S22 Ultra or the Redmi Note 11. Additionally, there were still issue such as the app crashing causing a bad model update, sometimes making the model irrecoverable, high battery consumption and exploding gradient. Recommendations for future research include addressing issues like exploding gradients and experiment with implementing more complex models. Future work could focus on creating an interactive interface to modify the model, hyperparameters, image processing and ability to change the reward function to make an interesting reinforcement learning teaching platform. Moreover, insights from the foundational experiments underscore the value of accessible RL projects, highlighting platforms like OpenBot and games like Pong as useful learning tools. Overall, this study contributes to making RL more accessible and comprehensible, potentially expanding its use in mobile computing and robotics.

Acknowledgements

I want to express my sincere thanks to my family and my partner for their great support throughout my academic journey. Your encouragement and understanding have been my pillars of strength.

I would also like to express my appreciation to my supervisor, Naveed Muhammad, for his positivity and support.

Additionally, I want to thank the University of Tartu and the Institute of Technology for making this learning experience possible.

Finally, I am very grateful for Matthias Müller, who generously donated 4 OpenBot platforms to the University and allowed this research to be possible.

This achievement would not have been possible without the collective support and encouragement from my loved ones and mentors. Thank you for being an integral part of this rewarding experience.

[illegible]

Bibliography

[1] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 09 2017.

[2] GSMA, "The state of mobile internet connectivity 2023." `https://tinyurl.com/4jtjj4rv`, 2023.

[3] N. Rottmann, N. Studt, F. Ernst, and E. Rueckert, "Ros-mobile: An android application for the robot operating system," 2020.

[4] S. Pedre, M. Nitsche, F. Pessacg, J. Caccavelli, and P. De Cristóforis, "Design of a multi-purpose low-cost mobile robot for research and education," 09 2014.

[5] "TensorFlow Lite: Machine Learning for Mobile and Embedded Devices." `https://www.tensorflow.org/lite`. last last accessed: 2024-05-20.

[6] PROJECTPRO, "15 python reinforcement learning project ideas for beginners." `https://www.projectpro.io/article/reinforcement-learning-projects-ideas-for-beginners-with-code/521`. last accessed: 2024-05-19.

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. second ed., 2018.

[8] E. L. Thorndike, *Animal Intelligence*. 1911.

[9] B. F. Skinner, *The Behavior of Organisms*. 1938.

[10] B. F. Skinner, *Science and Human Behavior*. 1953.

[11] R. Bellman, "On the theory of dynamic programming," *Proceedings of the National Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.

[12] R. Bellman, *Dynamic Programming and Markov Processes*. 1957.

[13] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[14] C. Watkins, "Learning from delayed rewards," 1989.

[15] R. Sutton, "Learning to predict by the method of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[16] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine Learning Proceedings 1990* (B. Porter and R. Mooney, eds.), pp. 216–224, San Francisco (CA): Morgan Kaufmann, 1990.

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

[18] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992.

[19] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," 2019.

[20] A. Raghu, M. Komorowski, I. Ahmed, L. Celi, P. Szolovits, and M. Ghassemi, "Deep reinforcement learning for sepsis treatment," 2017.

[21] G. Huang, X. Zhou, and Q. Song, "Deep reinforcement learning for portfolio management," 2022.

[22] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, "Learning to drive in a day," 2018.

[23] L. Lyu, Y. Shen, and S. Zhang, "The advance of reinforcement learning and deep reinforcement learning," in *2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA)*, pp. 644–648, 2022.

[24] P. D. CHRISTOPHER J.C.H. WATKINS, "Q-learning, technical note," *Kluwer Academic Publishers*, vol. 8, pp. 279–292, 1992.

[25] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec 1943.

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[27] M. Müller and V. Koltun, "Openbot: Turning smartphones into robots," 2021.

[28] Müller, Matthias and Koltun, Vladlen, "OpenBot." `https://github.com/isl-org/OpenBot`, 2020. last accessed: 2024-05-20.

[29] L. Gras and L. Glorieux, "Autonomous vehicles project: Openbot." `https://medium.com/@lgopenbot/autonomous-vehicles-project-openbot-aa806fd1829c`, 2022. last accessed: 2024-05-20.

[30] OpenCV, "OpenCV Releases." `https://opencv.org/releases/`. last accessed: 2024-05-20.

[31] L. Gras, “Github repository forked from official openbot repository.” https://github.com/Lilousarg/OpenBot, 2024. GitHub, last accessed: 2024-05-20.

[32] L. Gras, “Demonstration of agents playing carpole and pong.” https://youtu.be/eUyczcUNVhA, 2024. last accessed: 2024-05-20.

[33] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.

[34] P. Contributors, “Pytorch tutorials: Intermediate reinforcement q-learning.” https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. last accessed: 2024-05-20.

[35] O. Vedpathak, “Playing pong using reinforcement learning.” https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d, 2019. Medium, last accessed: 2024-05-20.

[36] O. Vedpathak, “Pong from pixels.” https://github.com/omkarv/pong-from-pixels, 2019. GitHub, last accessed: 2024-05-6.

[37] A. Karpathy, “Deep reinforcement learning: Pong from pixels.” https://karpathy.github.io/2016/05/31/rl/, 2016. Blog, last accessed: 2024-05-6.

[38] L. Gras, “Demonstration of reinforcement learning training on openbot at different stages.” https://youtu.be/BXwLrwt9fgc?si=U3hk7zhIC9bWNYsm, 2024. last accessed: 2024-05-20.

[39] NVIDIA, “CUDA v11.3.” https://developer.nvidia.com/cuda-11.3.0-download-archive?target_os=Linux&target_arch=x86_64&Distribution=Ubuntu&target_version=20.04&target_type=deb_network. Version 11.3, last accessed: 2023-12-21.

[40] NVIDIA, “CUDA installation guide.” https://docs.nvidia.com/cuda/pdf/CUDA_Installation_Guide_Windows.pdf. last accessed: 2023-12-21.

[41] Google for Developers, Android Developers, “Sensors overview.” https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview. last accessed: 2023-12-21.

[42] OpenAI, “ChatGPT: An ai language model by openai.” https://openai.com/chatgpt. last accessed: 2024-05-20.

Appendices

10.1 External Tools Used in the Making of this Thesis

Following is a list of tools and websites that helped in the research and writing of this thesis:

- **TensorFlow [5] library and websites:** There are lots of documentation on how to correctly use tensorflow and adapt it to a specific scenario. This was very important to understand the pre-existing scripts and how to implement the reinforcement learning part.
- **CUDA websites and online guides [39] [40]** Installing the correct version of CUDA can be challenging with all the possible resulting conflicts from installing a version incompatible with the different owned libraries. Online guides and the CUDA website helped tremendously and allowed to accelerate the training to up to 10 times faster.
- **Python's and programming on Android Studio [41]:** Luckily there are lots of documentation online to help programming in Python and JavaScript. They were crucial to the understanding of the current scripts as well as a great help to implement the new libraries, such as OpenCV on Android Studio.
- **ChatGPT [42]:** Chat GPT greatly helped with the debugging process in programming either in Python or Java by reading the long error logs and giving back the specific place in the code the error occured at. It can also specify what kind of error it was and what to look at. While not always reliable, it is a great tool to help with debugging scripts.

10.2 Implementing OpenCV on Android Studio

Here are the details for this installation:

1. **The installation of the correct version of OpenCV:** In this case, version 4.8.0 [30] was installed, followed by the extraction of the folder to the preferred path.
2. **The addition of the library to Android Studio:** Go to File > New > Import Module..., select the OpenCV folder and SDK, resulting in the creation of a source directory similar to `".../OpenCV-android-sdk/sdk"`. Rename the module as ”opencv”.
3. **Creating the namespace:** Find the gradle file from the OpenCV recently added module, in Android Studio it may appear in red and named `build.gradle (:opencv).` Add the namespace as follows:

```
    android {
        namespace 'org.opencv'
```

Normally, sync the gradle should work properly now.

4. **Adding the dependency:** Go to File > Project Structure. Select `Dependencies` then `app` followed by the + symbol under `declared dependencies`. Select `Module Dependency` then `opencv`. The dependency is now added to the app.

5. **Debugging the last errors:** in your `build.gradle (:opencv)`, add the following:

```
    android {
        namespace 'org.opencv'
        ...

        buildFeatures{
            aidl true
            buildConfig true
        }
    }
```

6. **Versioning:** The following versions were utilized in this experiment to ensure the smooth operation of the app without conflicts:

   - OpenCV : 4.8.0
   - in `build.gradle (:opencv)`:

```
        compileSdk 34


        defaultConfig {
            minSdkVersion 21
            targetSdkVersion 32
            ...
        }
        compileOptions {
            sourceCompatibility JavaVersion.VERSION_11
            targetCompatibility JavaVersion.VERSION_11
        }
```

   - in `build.gradle (:app)`:

```
        compileSdk 34

        defaultConfig {
            applicationId "org.openbot"
            minSdkVersion 21
            targetSdkVersion 32
            ...
        }
        compileOptions {
        coreLibraryDesugaringEnabled true

        sourceCompatibility = '11'
        targetCompatibility = '11'
        }
```

10.3 Forward and Backward Policy

Following are the functions implemented in $MyModel$ to compute the forward and backward pass:

```
public Object[] policyForward(RealMatrix x) {

        RealMatrix w1Matrix =
         ↪ MatrixUtils.createRealMatrix(model.get("W1"));
        RealMatrix xTranspose = x.transpose();
        RealMatrix hMatrix = xTranspose.multiply(w1Matrix);

        double[][] h = hMatrix.getData(); // Calculate the dot
         ↪ product

        h = relu(h); // Apply ReLU activation function

        hMatrix = MatrixUtils.createRealMatrix(h);
        RealMatrix w2Matrix =
         ↪ MatrixUtils.createRealMatrix(model.get("W2"));
        RealMatrix logProbMatrix = hMatrix.multiply(w2Matrix);
        double[][] logProb = logProbMatrix.getData(); // Calculate
         ↪ log probability


        double[] p = softmax(logProb[0]); // Calculate probability

        // Return probability and hidden state
        return new Object[]{p, h};
    }

```

```
public Map<String, double[][]> policyBackward(RealMatrix eph,
 ↪ RealMatrix epx, RealMatrix epdLogProb) {

        RealMatrix ephTranspose = eph.transpose();
        // Calculate gradients for W2
        RealMatrix dW2Matrix = ephTranspose.multiply(epdLogProb);

        double[][] dW2 = dW2Matrix.getData();

        // Calculate gradients for dh
        RealMatrix dhMatrix =
        ↪  epdLogProb.multiply(MatrixUtils.createRealMatrix(model.get("W2")).tr
        double[][] dh = dhMatrix.getData();

        // Apply backpropagation for ReLU activation
        for (int i = 0; i < eph.getRowDimension(); i++) {
            for (int j = 0; j < eph.getColumnDimension(); j++) {
                if (eph.getEntry(i,j) <= 0) {
                    dh[i][j] = 0; // Apply ReLU function
                }
            }
        }

        RealMatrix dhOriginal= MatrixUtils.createRealMatrix(dh);
        RealMatrix dhTranspose = dhOriginal.transpose();
        RealMatrix dW1Matrix = dhTranspose.multiply(epx);
        double[][] dW1 = dW1Matrix.transpose().getData();

        Map<String, double[][]> gradients = new HashMap<>();
        gradients.put("W1", dW1);
        gradients.put("W2", dW2);

        return gradients;
    }
```

Non-exclusive licence to reproduce thesis and make thesis public