Edge implementation of a passive policy for the control of a simple pendulum

Gianluca Centonze*, Riccardo Zanella

[1] University of Bologna, Italy.

[2] University of Twente, The Netherlands.

Corresponding author email: giangi03.111@gmail.com

January 25, 2025

Abstract

This work investigates the implementation of a passively learned policy for controlling a simple pendulum on a microcontroller. The policy was trained within a reinforcement learning framework in two stages: initially, the system was trained with unlimited energy to determine the energy required for the control task. A second training phase was performed with limited energy using the concept of a virtual energy tank to ensure the system's passivity. Once trained, a physical device based on a microcontroller was developed to control a permanent magnet DC motor, read an encoder, and infer the learned policy. The final goal was to evaluate the correspondence between simulation results and the behavior of the physical system, analyzing the feasibility and performance of the device in the real world. The results demonstrated how closely simulations can approximate reality and provided insights into the potential application of this approach to other passive control scenarios.

Contents

List of Figures

Chapter 1

Introduction

In recent years, the evolution of control technologies and artificial intelligence has led to significant advances in the field of automation. In particular, machine learning techniques, such as reinforcement learning, have developed considerably for controlling complex dynamic systems. A classic example is the control of a simple pendulum, a nonlinear system that requires sophisticated control techniques to ensure stable balancing. This work aims to address this challenge through the implementation of a passively learned policy executed on a microcontroller. Before delving into the technical and experimental details of the project, the fundamental concepts on which this work is based will be introduced, namely: the fundamentals of neural networks, reinforcement learning, and passivity in dynamic systems.

1.1 Neural Networks

Neural networks are one of the cornerstones of modern artificial intelligence and deep learning. Inspired by the structure and functioning of biological neurons, these networks are computational models designed to recognize patterns and make decisions based on data, whether simple or complex.

The first version of a neural network was presented by *Warren Sturgis McCulloch and Walter Pitts* in 1943, capable of approximating simple Boolean functions. Shortly after, neural networks called *Perceptrons* were developed, capable of recognizing and classifying simple patterns, thanks to the introduction of the concept of weights, which allowed the networks to "learn" from data. However, significant progress was made in the 1990s when increased computational capacity allowed the development of advanced learning algorithms capable of handling complex tasks. Over the years, various architectures for neural networks have been created, each designed to leverage specific properties of the data they process. Below, we refer to *multi-layer Perceptrons* (MLP), a variant of the Perceptron.

1.1.1 Structure

MLPs consist of several layers with a variable number of neurons; the outermost ones are input and output, while the inner ones are called *hidden layers*. Generally, all adjacent nodes are fully connected to each other.

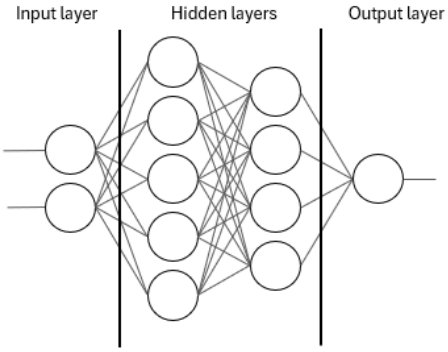


Figure 1.1: Fully connected feedforward neural network

The individual neuron, also called a node, is the basic unit of the neural network. It receives an input signal from one or more neurons or an external source (in the case of the input layer) and processes an output that goes to other neurons or to the outside (in the case of the output layer). To each input signal $x_i \in X$, the input vector, is associated a weight $w_i \in W$, the weight vector. The neuron simply performs a weighted sum of the inputs and adds a value called *bias*. The neuron's output is therefore represented by the equation:

$$y_i = \langle X, W \rangle + b = x_1 w_1 + x_2 w_2 \cdots x_n w_n + b \tag{1.1}$$

The addition of the bias allows for more versatility, such as the possibility of adding thresholds to the activation of a neuron.
Before the output signal reaches the next neuron or the outside, an *activation function* is added. There are several, and some of the most famous include ReLU 1.2 and sigmoid 1.3. The latter is very useful because it allows mapping the input signal within the range $[0, 1]$, preventing the summing of very high values.

$$ReLU(x) = max(0, x) \tag{1.2}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.3}$$

By adding the activation function, equation 1.1 becomes:

$$y_i = \sigma(x_1 w_1 + x_2 w_2 \cdots x_n w_n + b) \tag{1.4}$$

The learning process, therefore, consists of finding a set of weights and biases such that

the output function of the neural network can best approximate the desired function. The parameters to be updated are not manually manageable, so the training process is entirely done by the computer. The intermediate layers, called *hidden layers*, are so named because, not knowing exactly how processing occurs within them, only the inputs and the final effects on the outputs can be observed.

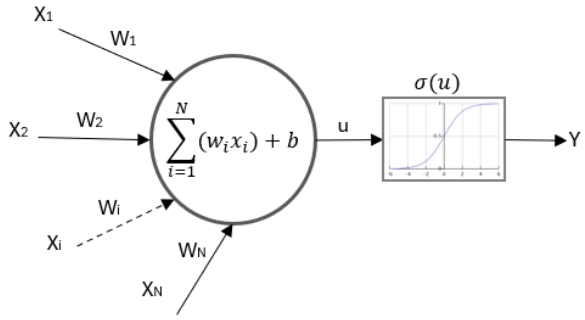


Figure 1.2: Inputs and outputs of the neuron

1.1.2 Training Process

The training process is an optimization problem. Given a set of inputs and corresponding outputs $\{x_k, y_k\}_{k=1...m}$, called the *training set*, the task is to find the set of parameters $\Theta^* \in \mathcal{R}^n$ such that a new input x_{m+1} is mapped by the neural network to the corresponding output y_{m+1}. In formula:

$$y_{m+1} \approx f(\Theta^*, x_{m+1}) \tag{1.5}$$

To quantify the error made by the neural network during training, the $loss function$ or RMSE (root mean squared error) loss function is defined:

$$L(\Theta) = \frac{1}{m}\sum_{k=1}^{m} ||y_k - f(\Theta, x_k)||^2 \tag{1.6}$$

The goal is to find Θ^* that minimizes the loss function:

$$\Theta^* = \underset{\Theta \in \mathcal{R}^n}{\arg\min} L(\Theta) \tag{1.7}$$

One of the optimization algorithms used is *Gradient Descent*. The basic idea is to update the parameters Θ in the opposite direction to the gradient of the loss function. The gradient is a vector of partial derivatives pointing in the direction of maximum growth of the function, so moving in the opposite direction seeks to reduce the loss function. The update of the parameters using Gradient Descent is given by the formula:

$$\Theta_{i+1} = \Theta_i - \alpha \nabla L(\Theta_i) \tag{1.8}$$

where α represents the step size to be taken in the opposite direction of the gradient.

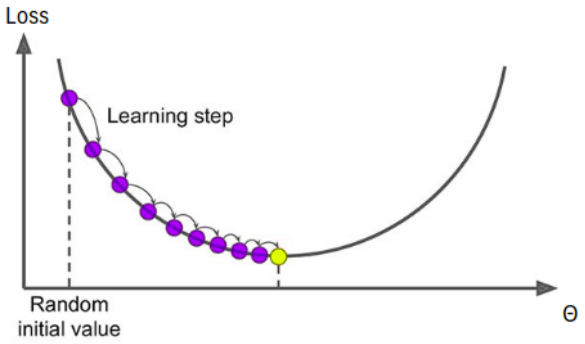


Figure 1.3: Visualization of Gradient Descent

1.2 Reinforcement learning

Reinforcement learning (RL) is a type of machine learning inspired by the way humans and other animals learn through interaction with their surrounding environment. Unlike supervised approaches, where a model is trained on a labeled dataset, reinforcement learning involves an agent that makes decisions in a context that is often uncertain and dynamic, without knowing in advance the best action to take. The agent interacts with the environment through a "trial and error" process: by observing the system's state, it takes an action that, if considered effective, will be more heavily weighted in future iterations. This method is particularly effective in situations where the environment is complex or lacks predefined rules. The agent does not need a teacher to provide the correct answers, but instead builds its own knowledge autonomously through accumulated experience, becoming increasingly skilled at taking actions that lead to positive outcomes.

1.2.1 Fundamental Definitions

AGENT

The agent is the entity that explores the environment and chooses the actions to take, aiming to improve its decisions over time through learning based on the feedback it receives from the environment.

ENVIRONMENT

The environment is everything that surrounds the agent and with which it interacts, providing new observations and rewards.

STATE

The state $s \in \mathcal{S}$ is the representation of the current situation of the environment.

ACTION

The action $a \in \mathcal{A}$ is what the agent decides to do in a given state.

REWARD

The reward $r \in \mathcal{R}$ is the feedback the agent receives from the environment after performing an action. The value of the reward can be positive or negative and reflects the quality of the action chosen by the agent.

POLICY

A policy π is a function that maps each state to an action. It can be deterministic 1.9 (a specific action for each state) or stochastic 1.10 (a probability distribution over possible actions).

$$a_t = \pi(s_t) \tag{1.9}$$

$$\pi(a_t|s_t) \tag{1.10}$$

OBSERVATION

The observation represents the state that the agent can use to make decisions; it may coincide with the state or be different.

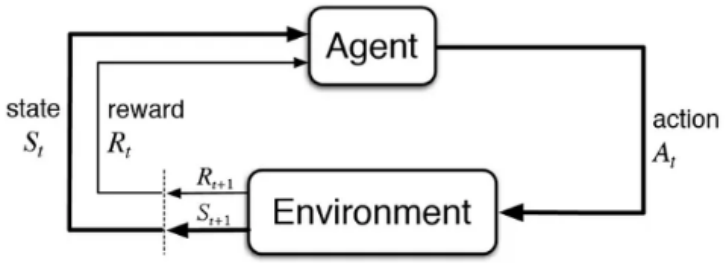


Figure 1.4: Reinforcement learning feedback

Based on the given definitions, there are two important functions in reinforcement learning:

1. **ACTION-STATE VALUE**
   It represents an estimate of the reward that can be obtained starting from state s by taking action a, assuming that the agent is following the policy π.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}\left[\sum_{t=t'}^{T} r(s_{t'}, a_{t'}) \middle| s_t, a_t\right] \tag{1.11}$$

2. **STATE VALUE**
   It represents an estimate of the reward that can be obtained from state s by following policy π.

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}\left[\sum_{t=t'}^{T} r(s_{t'}, a_{t'}) \middle| s_t\right] \tag{1.12}$$

1.2.2 Markov Property

As seen, the policy determines which actions to take based on the current state of the system. However, in theory, in order to decide which action to take, one would need to consider the entire past history of the system, which would be computationally expensive, if not impossible. To simplify, decisions need to be made by observing only the current state. In other words, the system's behavior must be "memoryless" such that all the necessary information to predict the next state is contained within the current state. A decision process of this kind is called a *Markov Decision Process* (MDP), mathematically described by the following relation:

$$P(s_{t+1}, r_{t+1}|s_t, a_t) = P(s_{t+1}, r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \cdots, s_0, a_0) \tag{1.13}$$

Most of the reinforcement learning theory deals with environments that satisfy the Markov property, as it guarantees convergence to an optimal solution.

1.2.3 Learning Methods

The goal of *reinforcement learning* is to find the optimal policy π^* that allows the agent to perform the desired task in the best possible way. To achieve this goal, the agent must balance two behaviors: *exploration*, which involves seeking new actions, and *exploitation*, which means using known actions that have led to good results. Effective balancing is crucial, as focusing too much on exploration may prevent the agent from obtaining immediate rewards, while focusing solely on exploitation risks limiting the discovery of better actions, thus blocking learning and performance improvement.

In reinforcement learning, there are two main approaches for finding the optimal policy π^*: gradient-based methods, value-based methods, and actor-critic methods.

GRADIENT-BASED METHODS

Gradient-based methods optimize the policy parameterized by a neural network. Learning occurs by updating the network's parameters to maximize the sum of future rewards the agent can obtain. These methods may suffer from high variance in results and require additional techniques to stabilize the learning process.

VALUE-BASED METHODS

In these approaches, learning is not directly about the policy but about a value function that provides an estimate of expected rewards. Once this function is learned, the optimal policy is obtained indirectly by selecting actions that maximize this estimate. These methods tend to be more stable but are less effective in particularly complex environments.

ACTOR-CRITIC METHODS

Actor-critic methods, currently the state of the art, combine the two previous approaches by exploiting the advantages of both. Specifically, a neural network is used to represent the policy (as in gradient-based methods) called the *actor*, while a value function, called the *critic*, is also learned to estimate the quality of actions (as in value-based methods). This combination is advantageous because it allows the actor to continually improve its policy through direct feedback, while the critic, by updating the value function, ensures that the learning process remains stable (see [12]).

1.3 System Passivity

1.3.1 Definition

Passivity is a fundamental concept in the theory of dynamic systems and control. It refers to a system's ability to accumulate and dissipate energy without actively producing it. A system is said to be passive if, regardless of the input type, it cannot generate more energy than it receives. This implies that a passive system cannot diverge uncontrollably. Mathematically, a system is considered passive if there exists a non-negative scalar storage (or energy) function $V(x)$ that satisfies, for every input $u(t)$ and output $y(t)$, the following inequality:

$$\dot{V}(x) \leq u(t)^T y(t) \tag{1.14}$$

where x represents the system's state. This inequality indicates that the change in the energy stored in the system cannot exceed the energy supplied to the system through input $u(t)$.

Passivity is closely related to the concept of stability, making passive systems particularly desirable in control applications where stability is a critical property. For example, in a robotic system, using control based on passivity can help ensure that the robot does not perform movements that lead to dangerous or unstable conditions.

In control theory, passivity is often exploited to design controllers that guarantee the stability of the closed-loop system. Passivity-Based Control (PBC) is an approach that utilizes the system's passivity to develop control laws that maintain or regulate the system's energy in a controlled way, thus ensuring the overall stability of the system.

1.3.2 Energy Tank

The concept of the *Energy tank* arises from the need to effectively manage energy in robotic systems. Energy tanks are useful mechanisms for implementing PBC, providing a way to store and regulate energy.
Mathematically, energy tanks can be described as dynamic systems that accumulate energy and transfer it in a controlled manner, ensuring that the robotic system remains passive. The dynamic model of an energy tank is described by the following equations:

$$\begin{cases} \dot{x} = u \\ y = \frac{\partial V(x)}{\partial x} \end{cases}$$

where x represents the state, u is the energy flow into the tank, y is the energy output available to the robot, and $V(x) = \frac{1}{2}x^2$ is the energy function that describes the stored energy inside the tank.

1.3.3 Connecting the Tank to a Dynamic System

The connection between the robot's dynamic system and the energy tank is made through an interconnection that preserves energy, meaning that no energy is created or dissipated during the exchange. This connection is represented by the following equation:

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 0 & P_{12} \\ P_{21} & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{1.15}$$

where u_1 and u_2 are the inputs of the two systems, y_1 and y_2 are the outputs of the two systems, and $P_{ij,i\neq j}$ represents the energy transfer function between the two systems. The interconnection matrix preserves power because the energy entering one subsystem is exactly equal to the energy leaving the other, with no losses or generation of energy.
The connection parameters are defined as follows:

$$P_{12} = \frac{w}{\partial V(x)/\partial x} \text{ and } P_{21} = -\frac{w^T}{\partial V(x)/\partial x}$$

The final connection becomes:

$$\begin{pmatrix} \tau \\ u \end{pmatrix} = \begin{pmatrix} 0 & \frac{w}{\partial V(x)/\partial x} \\ -\frac{w^T}{\partial V(x)/\partial x} & 0 \end{pmatrix} \begin{pmatrix} \dot{q} \\ y \end{pmatrix} \tag{1.16}$$

Equivalently written as:

$$\begin{cases} \tau = \frac{w}{\partial V(x)/\partial x} y = \frac{w}{\partial V(x)/\partial x} \cdot \partial V(x)/\partial x = w \\ u = -\frac{w^T}{\partial V(x)/\partial x} \dot{q} \end{cases} \tag{1.17}$$

where τ represents the control torque applied to the robot and w is the control input. However, when the term $\frac{\partial V(x)}{\partial x}$ becomes zero, the matrix enters a singularity condition, making it impossible to provide further control inputs. Thus, when the tank is empty, it can no longer transfer energy to the robot, effectively implementing the following control law:

$$w_{effective} = \begin{cases} w & V(x) \geq e^* \geq 0 \\ 0 & V(x) \leq e^* \end{cases} \tag{1.18}$$

for some small value of e^*, the energy required to complete the control action (for more information, refer to [16] and [2]).

Chapter 2

Creating the Environment

Before training a RL agent, it is necessary to create an environment with which it can interact. It is possible to build an environment in reality by creating a physical device, or to create a simulated one.
Initially, the model was calculated to define the rules by which the environment behaves, and subsequently, the iterative algorithm that manages the exploration of the agent.

2.1 Model

To create the environment that the agent will explore, the dynamic model of the pendulum, calculated using the Euler-Lagrange equations, was used. A representation as close as possible to the real model is essential to ensure a functional implementation on a microcontroller. For this purpose, the pendulum model shown in Figure was chosen.

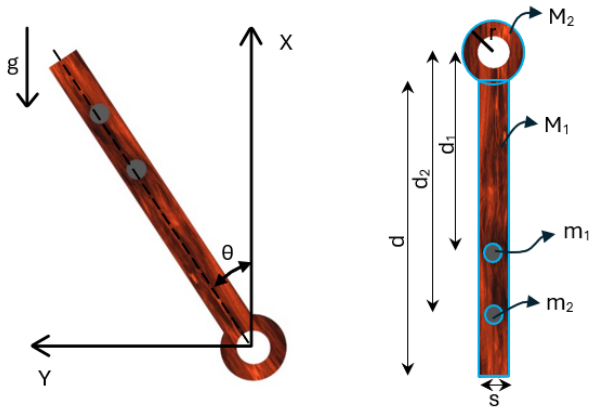


Figure 2.1: Pendulum used in the real model

2.1.1 Parameters

Figure 2.1 shows the actual shape of the pendulum with the geometric parameters and the various masses along with the reference system. The pendulum, to be analyzed, is

broken down into the fundamental geometries highlighted in blue: a parallelepiped for the rod, a hollow disk for the attachment to the motor shaft, and two point masses for the weights. The meanings of the various parameters used are defined as follows:

- d and s represent the length and width of the parallelepiped, respectively.
- d_1 and d_2 represent the distances of the two weights from the center of rotation of the pendulum.
- r represents the outer radius of the hollow disk, while the inner radius is $r/2$.
- M_1 and M_2 are the masses of the parallelepiped and the hollow disk, respectively.
- m_1 and m_2 are the masses of the weights placed on the pendulum.

The values of the various parameters are shown in table **??**.

2.1.2 Euler-Lagrange Equation

The starting point consists of calculating the potential energy U and kinetic energy K of the pendulum:

$$U = [M_1(\frac{d}{2} + r) + m_1 d_1 + m_2 d_2]g\cos\theta =$$

$$U = Mg\cos\theta \tag{2.1}$$

$$K = \frac{1}{2}I_{disk}\dot{\theta}^2 + \frac{1}{2}I_{rod}\dot{\theta}^2 + \frac{1}{2}m_1 d_1^2\dot{\theta}^2 + \frac{1}{2}m_2 d_2^2\dot{\theta}^2 =$$

$$= \frac{1}{2}(I_{disk} + I_{rod} + m_1 d_1^2 + m_2 d_2^2)\dot{\theta}^2 =$$

$$K = \frac{1}{2}I_{tot}\dot{\theta}^2 \tag{2.2}$$

To calculate I_{tot}, the inertia of each part must be considered. We have considered: a hollow disk rotating around its center, a parallelepiped rotating around a point at a distance r from its shortest end, and finally, two point masses acting as weights placed on the pendulum.

- $I_{disk} = \frac{1}{2}M_2(r^2 + \frac{r}{2})$ inertia of the hollow disk.
- $I_{rod} = \frac{1}{12}M_1(d^2 + s^2) + M_1(\frac{d}{2} + r)^2$ inertia of the parallelepiped.
- $I_{m_1} = m_1 d_1^2$ inertia of the point mass of m_1.
- $I_{m_2} = m_2 d_2^2$ inertia of the point mass of m_2.

From these expressions, the following is derived:

$$I_{tot} = I_{disk} + I_{rod} + I_{m_1} + I_{m_2} =$$

$$= \frac{1}{2}M_2(r^2 + \frac{r}{2}) + M_1(\frac{d^2}{3} + \frac{s^2}{12} + r^2 + dr) + m_1 d_1^2 + m_2 d_2^2 \tag{2.3}$$

Once the necessary equations are obtained, the dynamic model can be derived using the Euler-Lagrange equations, using the Lagrangian $\mathcal{L} = K - U$ and the generalized force of Lagrange $Q = u - b\dot{\theta}$, where u is the applied torque term, and $b\dot{\theta}$ represents the damping term.

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}}\right) - \frac{\partial \mathcal{L}}{\partial \theta} = Q \tag{2.4}$$

Since the kinetic energy is a function only of $\dot{\theta}$ and the potential energy is only a function of θ, we can rewrite the previous equation as:

$$\frac{d}{dt}\left(\frac{\partial K}{\partial \dot{\theta}}\right) + \frac{\partial U}{\partial \theta} + b\dot{\theta} = u \tag{2.5}$$

By calculating the derivatives, the final model is obtained:

$$I_{tot}\ddot{\theta} = u - b\dot{\theta} + Mg\sin\theta \tag{2.6}$$

2.1.3 Motor Static Model

In addition to the pendulum model, a model of a permanent magnet DC motor was also implemented, using the static equations. This choice is based on the observation that, during sampling intervals, the electrical transient is considered finished.

- V_a, i_a, r_a are the voltage, current, and resistance of the rotor or armature, respectively.
- u is the torque produced by the motor.
- ω is the angular velocity of the rotor.
- K_t, K_v are the torque and speed constants, respectively. They express the current-torque and speed-fcem ratio. The estimated motor parameters are listed in table **??**

The armature current i_a can be expressed as:

$$i_a = \frac{V_a - K_v\omega}{r_a} \tag{2.7}$$

From this relation, the control torque applied to the motor is calculated as:

$$u = i_a K_t \tag{2.8}$$

2.2 Custom Environment in Gymnasium

Gymnasium [5], is a library that provides a standard API for creating virtual environments useful for training in the field of reinforcement learning. It involves the creation of three main functions: *reset*, *step*, and *render*.
The *reset* function is called at the beginning of each episode and initializes the system's state and related variables, returning the initial observation of the environment. The *step* function updates the environment based on the action chosen by the agent, updating the system's state, calculating the reward obtained, and returning the next observation, along with two flags: `terminated` and `truncated`. These flags indicate respectively the end of the episode (termination) or forced interruption due to time limits (truncation). The *render* function allows for writing code to visualize the simulated system [6].

Algorithm 1 Simulating agent-environment interaction

Given: *an environment and a policy* π
$obs \leftarrow reset()$
for `K steps` **do**
  $action \leftarrow \pi(obs)$
  $obs, reward, terminated, truncated \leftarrow step(action)$
  $render()$
  if terminated or truncated **then**
    Break
  end if
end for

2.2.1 Reset Function

The reset function initializes the state $[\theta, \dot{\theta}]$ with random values chosen from the interval $[-1, 1]$, which allows the agent to explore different initial configurations. Additionally, the energy of the tank is restored, and the number of episodes performed is incremented. The observation returned to the agent is in the form:

$$obs = [\cos\theta, \sin\theta, \tanh\dot{\theta}] \tag{2.9}$$

An observation coinciding with the state of the system was not chosen because normalized values are better managed by the agent.

2.2.2 Step Function

The *step* function updates the system state by solving the pendulum's differential equation using the Euler numerical method and updating the remaining energy in the tank.

Algorithm 2 Euler Method for the Numerical Integration of the Pendulum Model

Input: Function $f(t, \dot{\theta}, \theta)$ dynamic model of the pendulum, step dt.
Output: Approximation of the solution at t_f
$\dot{\theta} \leftarrow \dot{\theta}_0$
$\theta \leftarrow \theta_0$
for `N steps` **do**
  $\dot{\theta} \leftarrow \dot{\theta} + dt \cdot f(t, \dot{\theta}, \theta)$
  $\theta \leftarrow \theta + dt \cdot \dot{\theta}$
end for
return $\theta, \dot{\theta}$

A too large dt could introduce numerical errors, so it is necessary to choose a sufficiently small dt, balancing the desired accuracy with the computational load. To this end, the chosen dt is $500ns$.

2.2.3 Reinforcement Learning and Energy Tank

The integration between RL and energy tanks represents an approach that aims to combine the versatility of reinforcement learning with the properties of passivity. The control policy can be passivized in inference by imposing a limit e^* equal to the energy required to complete the control action. Alternatively, as in the case study, the policy can be learned passively by setting a limit on the energy that can be used during the training phase.
In a discrete sampling interval, the energy leaving the tank can be described as:

$$\int_{T_k}^{T^{(k+1)}} -\dot{V}(x)\, dx \tag{2.10}$$

The energy leaving the tank during each integration period is the energy produced by the actuator $\tau\dot{q} = w\dot{q}$. Therefore, we can write:

$$\int_{T_k}^{T_{(k+1)}} -\dot{V}(x)\, dx = \int_{T_k}^{T_{(k+1)}} w^T(x)\dot{q(x)}\, dx \tag{2.11}$$

Considering constant control input throughout the integration period, we obtain the relation:

$$w_k^T \int_{T_k}^{T_{(k+1)}} \dot{q}(x)\, dx = w_k^T \left(q_{k+1} - q_k\right) := \Delta e_{k+1} \tag{2.12}$$

The energy released from the tank is therefore:

$$\Delta e_{k+1} = \begin{cases} w_k^T \left(q_{k+1} - q_k\right) & w_k^T \left(q_{k+1} - q_k\right) > 0 \\ 0 & w_k^T \left(q_{k+1} - q_k\right) \leq 0 \end{cases} \tag{2.13}$$

This definition ensures that energy is not reintroduced into the tank, which can only empty during the control action.

2.2.4 Reward Function

By adding the energy tank, the environment no longer satisfies the Markov property, because, in the case of the tank being emptied, the future dynamics of the system are influenced by past states. To prevent this, when the tank empties, the episode is terminated via the *terminated* flag, resulting in a lower reward than if the tank never emptied. As a result, the reward function must necessarily be positive to prevent the agent from terminating the episode prematurely and collecting fewer negative rewards [16]. The reward function used is:

$$\left[1 + |pos\_error| + 0.1 \left|\tanh \dot{\theta}\right| + 0.01 \left|V_a\right|\right]^{-1} \tag{2.14}$$

The maximum reward obtainable at each step is one, which is assigned when the pendulum is stationary, no control is applied, and the pendulum is within the acceptable position error interval (pos_err), calculated by normalizing the angle θ in the interval $[-\pi, \pi]$ and applying the following law:

$$pos\_err = \begin{cases} 0 & l_1 \leq \theta \leq l_2 \\ \theta m_1 + q_1 & l_2 \leq \theta \leq \pi \\ -\theta m_2 + q_2 & -\pi \leq \theta \leq l_1 \end{cases} \tag{2.15}$$

This way, the error is zero when the pendulum is in the static friction range that allows it to stay almost upright without falling, even in the absence of torque from the motor. l_1 and l_2 represent the left and right boundary angles of zero, while m_1, q_1, m_2, and q_2 are the coefficients of the lines that map $\theta \in [-\pi, l_1]$ and $\theta \in [l_2, \pi]$ to the intervals $[-1, 0]$ and $[0, 1]$, respectively.

This solution was adopted to address the issue of static friction in the real model, caused by the brushes of the DC motor. This friction prevents precise control near the vertical position, causing instability. For this reason, the goal was set on an experimentally determined angle range that allows the pendulum to remain in balance by exploiting the friction itself.

Algorithm 3 Step Function

Input: $e = e_{init}$, $\theta, \theta' = \theta_0$, $\dot{\theta} = \dot{\theta}_0$, and $u, u' = 0$
for `M steps` **do**
  $u \leftarrow$ *action predicted by the policy*
  $u' \leftarrow u$
  $\theta' \leftarrow \theta$
  for `N steps` **do**
    $\dot{\theta} \leftarrow \dot{\theta} + dt \cdot f(t, \dot{\theta}, \theta)$
    $\theta \leftarrow \theta + dt \cdot \dot{\theta}$
  end for
  $r \leftarrow$ *reward*
  if $u'(\theta - \theta') > 0$ **then**
    $\Delta e \leftarrow u'(\theta - \theta')$
    $e \leftarrow e - \Delta e$
  end if
  if e <= 0 **then**
    return *terminated*
  end if
end for
return *truncated*

The parameter M represents the number of steps in an episode, and N the number of substeps needed to calculate the pendulum's position and velocity. In this way, the real behavior is simulated, where the pendulum's movement is continuous while the information is read at discrete times. The control and tank update frequency is determined by $(dt \cdot N)^{-1}$.

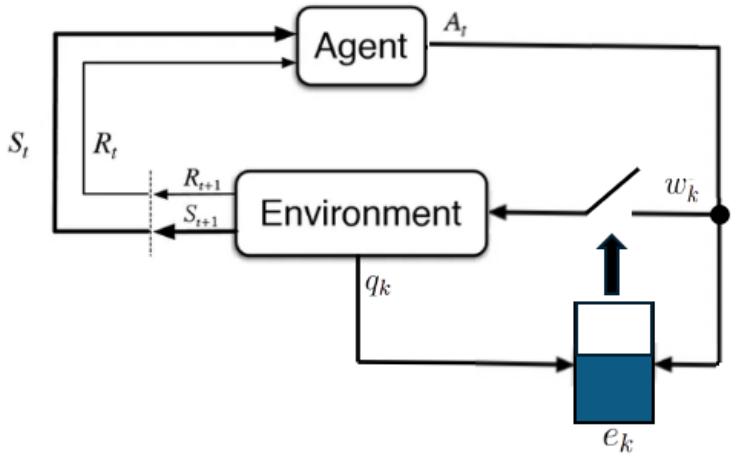


Figure 2.2: Reinforcement learning with passivization using an energy tank

2.2.5 Render Function

Visualizing the system, along with analyzing the graphs, is essential for evaluating the behavior of the policy during or after training. To this end, the *Pygame* library was used, which provides a simple way to create windows and draw polygons. An important aspect to consider is optimizing the code to avoid unnecessary overhead. For example, it is advisable to update only the areas of the screen that change with each call to the *render* function. A version of the code can be found in [9], and the result is shown in the figure:

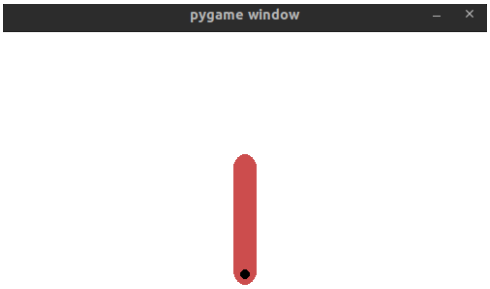


Figure 2.3: Pygame Window

Chapter 3

Training

3.1 Algorithm Setup

The training algorithm was implemented using the library provided by *Stable-Baselines3* [14]. It provides a set of learning algorithms, each belonging to one of three main categories: value-based, gradient-based, and Actor-Critic methods. For training the model, *Soft Actor-Critic* (SAC) was chosen with the default parameters provided by the library, with the addition of generalized State-dependent exploration (gSDE). SAC, in addition to having the advantages described in Chapter 1, can handle continuous states and actions. All simulations were carried out using an NVIDIA GeForce MX450 with an Intel i7-1165G7 processor clocked at 2.80GHz.

Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
ARS [1]	✔	✔	✘	✘	✔
A2C	✔	✔	✔	✔	✔
DDPG	✔	✘	✘	✘	✔
DQN	✘	✔	✘	✘	✔
HER	✔	✔	✘	✘	✔
PPO	✔	✔	✔	✔	✔
QR-DQN [1]	✘	✔	✘	✘	✔
RecurrentPPO [1]	✔	✔	✔	✔	✔
SAC	✔	✘	✘	✘	✔
TD3	✔	✘	✘	✘	✔
TQC [1]	✔	✘	✘	✘	✔
TRPO [1]	✔	✔	✔	✔	✔
Maskable PPO [1]	✘	✔	✔	✔	✔

Figure 3.1: Algorithms provided by stable baselines [15]

The training was conducted in a simulated environment, developed using Gymnasium (see Chapter 2), with the reward function designed to encourage the agent to bring the pendulum to a position sufficiently close to vertical. Two training sessions were carried out: one with unlimited energy and the other with limited energy, always maintaining

the same environment and a maximum number of 2500 steps per episode, to ensure a fair evaluation of performance.
The table 3.1 lists the notations used in the following sections.

3.2 Training with Infinite Energy

In the first training session, the agent had access to unlimited energy e_{inf}. The goal of this phase was to observe the agent's behavior without the restriction imposed by the energy tank. It is free to use any amount of energy to stabilize the pendulum, allowing it to explore more aggressive control strategies.

During the early phases of training, the agent tends to use large amounts of energy while freely exploring possible actions. However, as learning progresses, the agent begins to learn how to complete the desired control task, using only the necessary energy to perform the task. The agent's performance in terms of reward and energy used per episode is shown in graphs 3.4 and 3.3.

Once trained, the policy $\pi_{e_{inf}}$ was put through an evaluation phase: considering the deterministic policy, it was evaluated, over 200 episodes of 2500 steps each, on how much energy the agent consumes to complete the learned task.

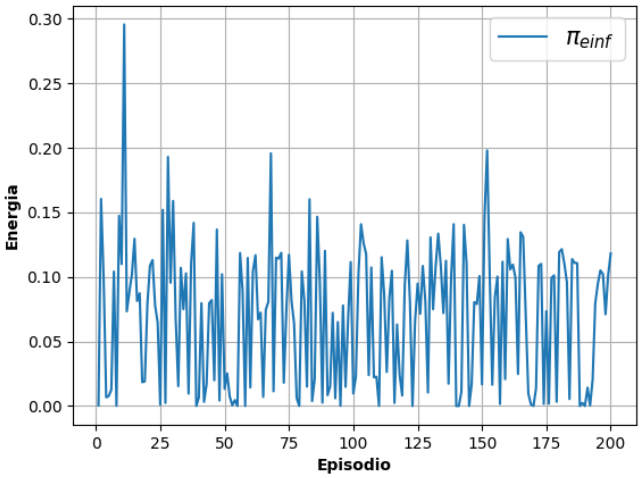


Figure 3.2: Energy consumed during evaluation by the policy trained with infinite energy

It can be seen that the values do not exceed $0.2J$, except for a peak, so it can be assumed that the energy e^*, necessary to successfully complete the control action, is about $0.2J$.

e_{inf}	unlimited initial energy in the tank
e^*	limited initial energy in the tank
$\pi_{e_{inf}}$	Policy trained with e_{inf}
π_{e^*}	Policy trained with e^*

Table 3.1: Notation used in the graphs

3.3 Training with Limited Energy

In the second training session, the agent was trained using the *energy tank* initialized to a value $e_0 = e^*$, then depleted according to the torque applied according to the law shown in algorithm [3]. If the remaining energy drops below a preset threshold, the episode is terminated.

This configuration requires the agent to develop more energy-efficient strategies, balancing energy use with the need to maintain the pendulum's stability.

As seen in graphs 3.4 and 3.3, training the agent with energy limitations π_{e^*} required more steps to complete because early termination of the episode disturbed the process, as shown in Figure 3.5. However, the spikes of energy consumed at the beginning of the training were avoided, thus preventing dangerous behavior, especially when training is conducted on a real device.

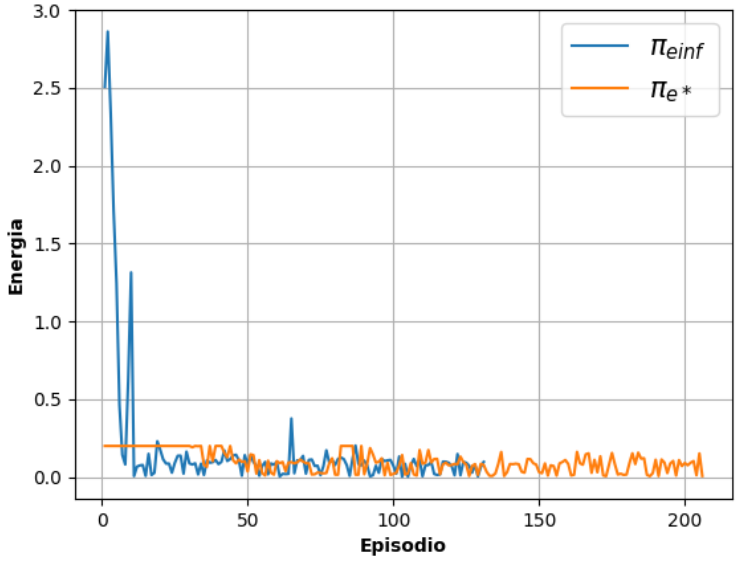


Figure 3.3: Energy used in training by the two agents

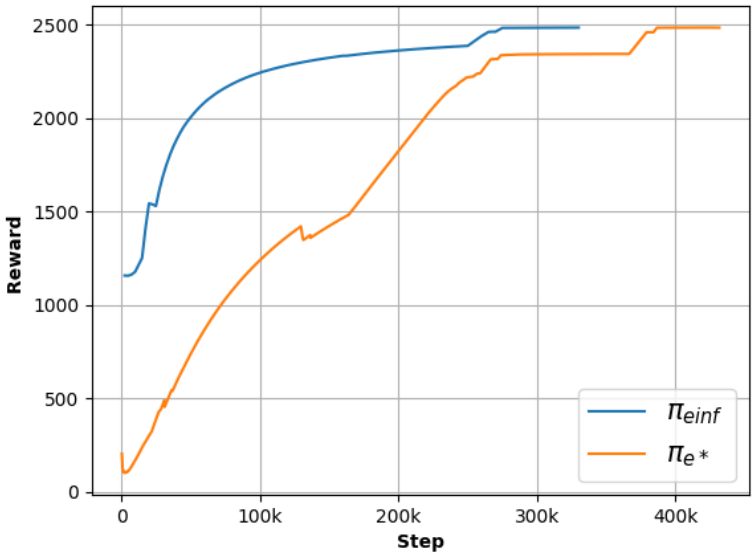


Figure 3.4: Reward obtained during training

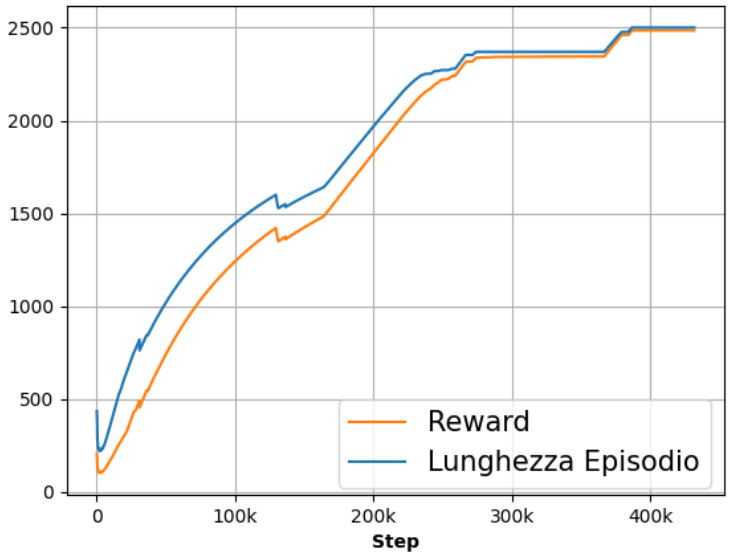


Figure 3.5: This graph shows how the reward obtained by the agent is proportional to the length of the episodes, causing it to consume less energy and thus avoid episode termination.

3.4 Inference Performance

After training, both configurations were evaluated during inference over 200 episodes of 500 steps each. Performance was measured in terms of the average position error (graph 3.6) and the average energy consumption per step (graph 3.7).

In graph 3.6, the progression of the average position error (pos_err, see Chapter 2) per

step for both trained policies $\pi_{e_{inf}}$ and π_{e^*} is shown. This progression highlights how the model is able to quickly learn an effective control strategy in both configurations. In fact, both agents manage to stabilize the pendulum vertically in about 50 steps, with some differences. It can be observed that, in the case of the agent π_{e^*}, the trajectory converges, albeit slightly slower, to stability, showing a more gradual action. Additionally, the agent $\pi_{e_{inf}}$ has small oscillations after reaching an almost null error, due to a more "nervous" control action.

Regarding energy usage, analyzing graph 3.7, it is clear that energy use is significantly different between the two configurations. The agent $\pi_{e_{inf}}$ uses more energy at each step, with the curve stabilizing around a value of about $0.08J$. This behavior reflects the fact that the agent has no energy constraints and can adopt more aggressive strategies to stabilize the pendulum. On the other hand, the agent trained with the energy tank constraint shows much lower energy consumption, with the curve stabilizing around $0.06J$. This demonstrates how the agent π_{e^*} learned to balance energy usage and the pendulum's stability, adopting more energy-efficient strategies.

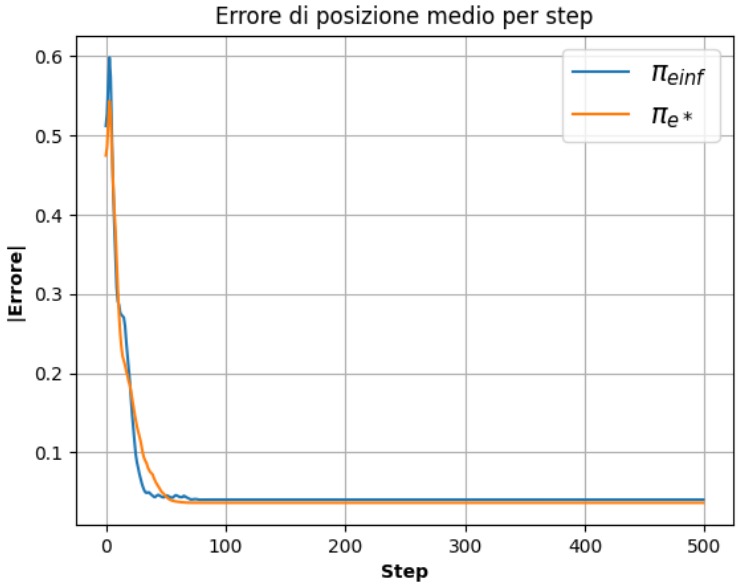


Figure 3.6

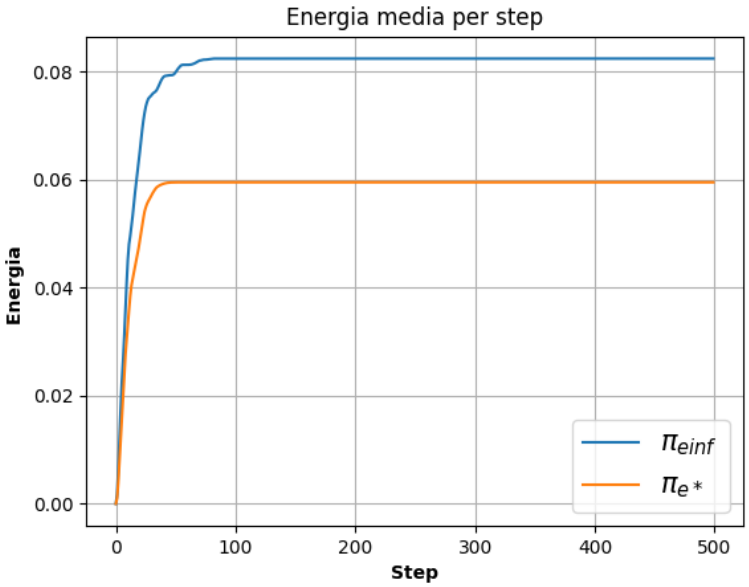

Energia media per step
Energia
Step
0.00
0.02
0.04
0.06
0.08
0
100
200
300
400
500
π_{einf}
π_{e*}

Figure 3.7

Chapter 4

Implementation on Microcontroller

4.1 Creation of the Physical Model

4.1.1 Electronic Components

DC MOTOR DRIVER

To power the DC motor, the *LM298N* driver was used, known for its cost-effectiveness and versatility. It is a circuit integrating a dual H-bridge, enabling bidirectional control of two DC motors or one stepper motor. In our case, the two terminals of the motor

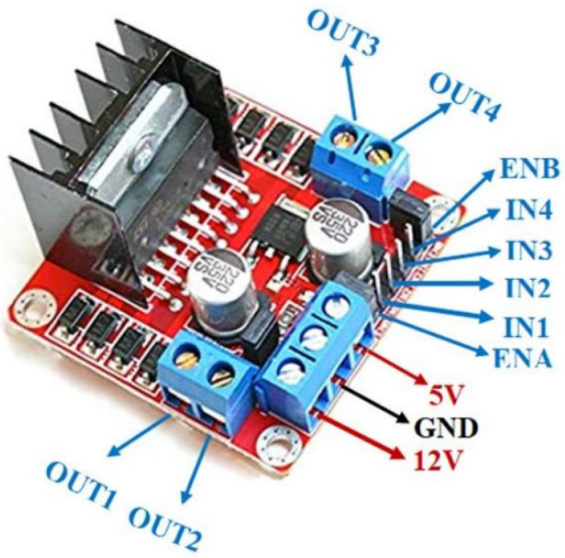


Figure 4.1: Driver with power and control ports

were connected to the OUT1 and OUT2 ports, while the IN1 and IN2 inputs received PWM signals from the microcontroller, allowing the regulation of the motor's speed and direction. [8]

BUCK DC CONVERTER

To simplify the power supply system, only one power source was used. However, the microcontroller requires 5V, whereas other components operate at 12V. For this purpose, the LM2596 DC converter was used, efficiently reducing the voltage from 12V to 5V, ensuring compatibility among the devices.

Figure 4.2: DC Converter

INCREMENTAL ROTARY ENCODER

Position measurement was carried out using an incremental rotary encoder with a resolution of 500 steps per revolution. However, this resolution was insufficient for precise control, so a quarter-step reading was implemented, resulting in an effective resolution of 2000 steps per revolution.

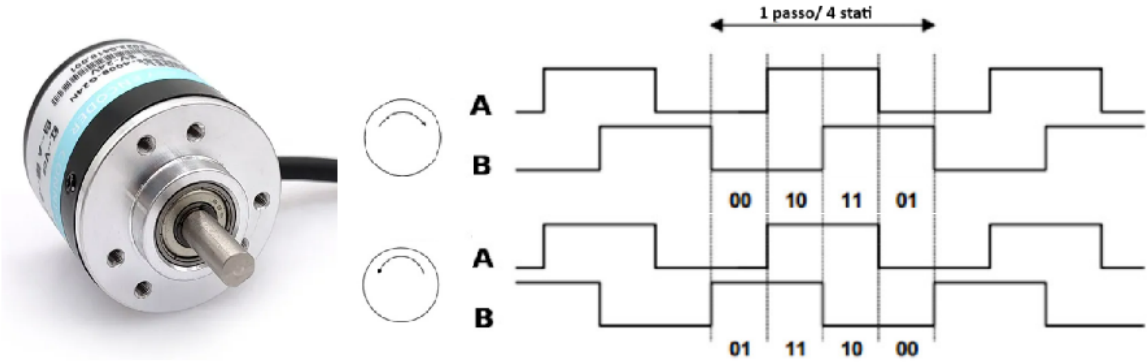


Figure 4.3: Encoder reading logic with quarter-step reading

The encoder used lacks a zero signal; thus, when the microcontroller is powered on, the pendulum must be at a known initial angle to correctly initialize the position. An elastic coupling was used to join the motor shaft with the encoder shaft, accommodating slight misalignments. [10]

MICROCONTROLLER

The device managing all components and the model inference is the *Esp32 Wroom 32* microcontroller. It features a dual-core processor with a clock frequency ranging from 80

to 240MHz, alongside 520KB of RAM and 448KB of ROM. These characteristics make the Esp32 a suitable candidate for handling the model inference, which demands substantial RAM. The dual-core configuration is particularly beneficial for task separation: one core manages the motor and encoder reading, while the other executes the neural network inference, reducing processing times. [3]

Figure 4.4: Esp32 Wroom 32

MOTOR

The motor used to drive the pendulum is a low-end DC motor with permanent magnets, repurposed from other applications. The estimated electrical and mechanical parameters are listed in the table below.

V_a	$\approx$	$12V$
r_a	$\approx$	30Ω
K_t	$\approx$	$0.07Nm$

Table 4.1

4.1.2 Mechanical Components

PENDULUM

The pendulum was made by gluing two identical shapes cut with a laser from a 5mm plywood board. The hole at the pendulum's end is slightly larger than the motor shaft. To ensure a secure connection between the two components, a pin was added, which, when screwed in, provides friction on the shaft, maintaining the pendulum's alignment during rotation.

Figure 4.5: Connection of the pendulum to the motor shaft

d	190mm
d_1	120mm
d_2	160mm
s	20mm
M_1	10g
M_2	1.5g
m_1	5g
m_2	7g

Table 4.2: Pendulum parameters referring to image **??**

SUPPORT BASE The support base is made of iron sheet and designed to hold the motor aligned with the encoder. It also features space for securing the perforated plate where all the electronic components are welded. In the drawing 4.6, the base is shown at the bottom, with rectangular holes where the support tabs for the motor and encoder will be welded. The dimension markings adjacent to the components indicate that a 90° bend should be made during sheet processing.

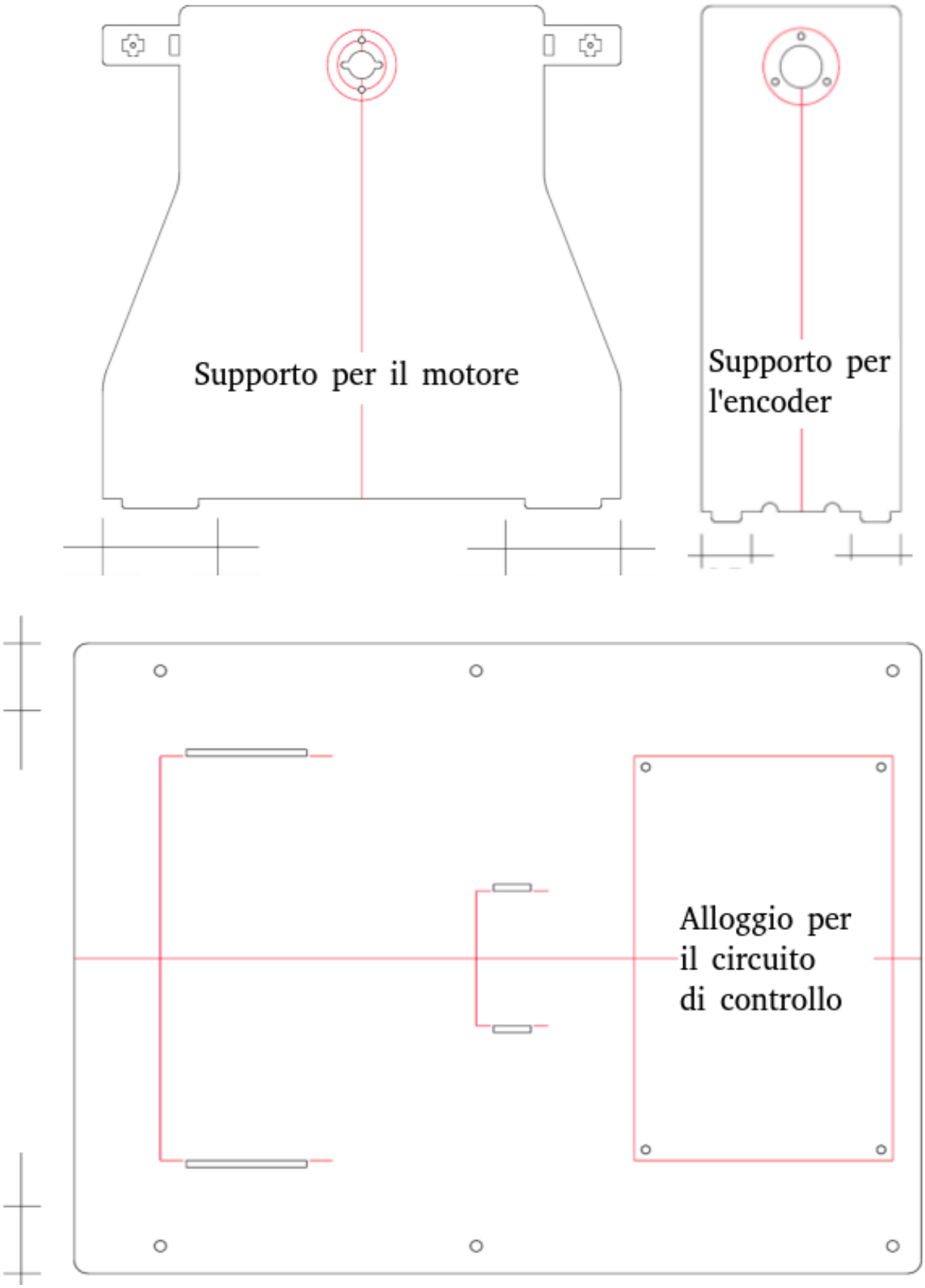


Figure 4.6: CAD drawing of the base

4.1.3 Control Circuit Diagram

1

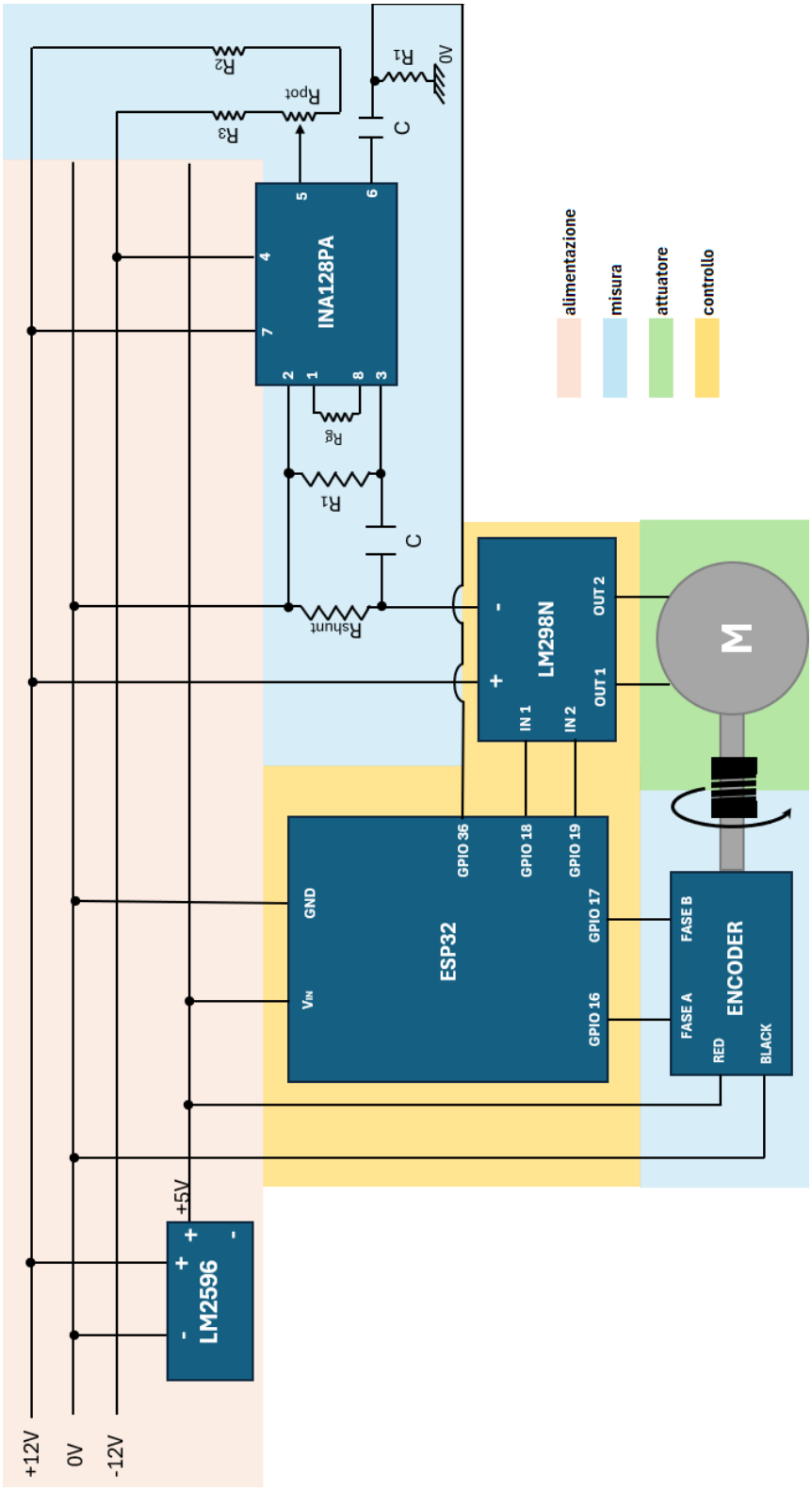
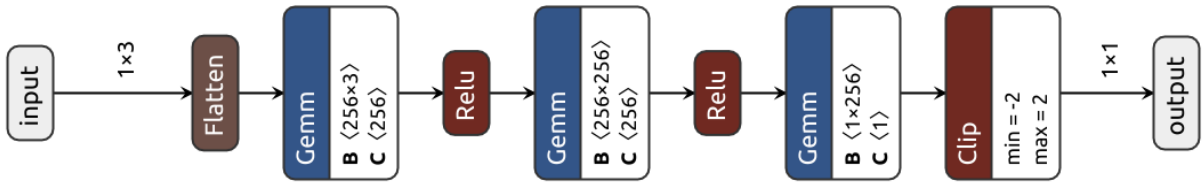


Figure 4.7: Circuit diagram implemented to actuate the pendulum

[1]For the part including the INA, refer to Appendix A

4.2 Neural Network Conversion

Once the circuit controlling the real devices has been designed, it is necessary to convert the neural network of the agent into a format capable of being executed on a microcontroller. From the networks used for training, only the actor's network needs to be exported, as the best policy has already been learned during training and the feedback from the critic is no longer needed.
The first conversion was done by exporting the *Stable-Baselines3* model in the *.onnx* format [13], which is supported by many RL frameworks.



Rete neurale dell'actor]fig:Rete neurale dell'actor

Figure 4.8: Image obtained via the web app Netron [11]

The image above shows the structure of the neural network after training, converted to ONNX format. The network has one hidden layer with 256 neurons, an input layer with 3 neurons corresponding to the observation size, and an output layer with a single neuron representing the action space size. Additionally, a clipping block was implemented at the network's output, introduced during the design phase with Stable-Baselines3, ensuring that the output is limited to the range [-2, 2]. However, our range of interest is [-1, 1], so further clipping will be required directly on the microcontroller to achieve the desired range.

Next, the Edge Impulse platform [7] was used, which allows uploading the trained model and converting it into a C++ library. This platform also enables testing the model by providing inputs and observing the generated output. Additionally, it estimates the execution time of the model, with the option to select from common devices for realistic performance simulations.

4.3 ESP-IDF Setup and Code

The ESP32 microcontroller was programmed using ESP-IDF, the development environment provided by *Espressif*, the manufacturer of the board. The folder structure and CMake files used to compile the project were adapted from the GitHub repository [1], which provides an example for standalone inference on ESP32 using ESP-IDF. This sped

up the integration of the library and simplified the development environment setup.

The written code performs several tasks: on one core, data acquisition from the encoder, motor control, and sending debug data via UART are executed, while the neural network runs on the other core. The data is then sent via UART to a Python script, which recognizes the input and logs the received values in a CSV file.

The script below includes all the peripheral configuration functions and the relevant interrupts for the encoder. It also contains the creation of the real-time inference task managed by FreeRTOS. The official API was referenced for configuring each peripheral, which can be found in [4].

```
extern "C" void app_main(void) //extern "C", because the C++ compiler throws an error on
    the app_main() function otherwise
{
    ESP_LOGI(LOG_TAG, "SETUP!");

    encoder_config();

    motor_pwm_config();

    gpio_install_isr_service(0);
    gpio_isr_handler_add(PHASE_A, phase_a_isr_handler, (void*) GPIO_NUM_16);
    gpio_isr_handler_add(PHASE_B, phase_b_isr_handler, (void*) GPIO_NUM_17);

    xTaskCreatePinnedToCore(inference_task, "Inference Task", 4096, NULL, 5, NULL, 1);
}
```

The following shows the code of the task performing inference, which, once called, repeats indefinitely due to the outer loop. The inner loop manages the episode system, each consisting of 200 steps, until the energy in the tank is depleted.

```
void inference_task(void *arg)
{
    int stp, ep, tot_steps = 0;
    int64_t inference_start_time, inference_end_time, inference_duration;
    float th, vel, prev_th, de, tank, delta_theta, pos_err;
    float best_action, prev_best_action, dt, u = 0;
    uint16_t dutyCycle;
    signal_t obs;
    ei_impulse_result_t result;
    EI_IMPULSE_ERROR resp;
    while (1)
    {
        mcpwm_comparator_set_compare_value(comparator, 0);
        printf("Place the pendulum\n");
        vTaskDelay(500);
        set_angle(180.0);
        th, vel, de, prev_th, stp = 0;
        tank = TANK_INIT;
        ep++;
        while (stp <= MAX_STP)
        {
            //////////////MODEL PREDICTION ///////////
```

```
            // functions required by the Edge Impulse API
            features[0] = cos(th);
            features[1] = sin(th);
            features[2] = tanh(vel);
            obs.total_length = 3;
            obs.get_data = &raw_feature_get_data;
            resp = run_classifier(&obs, &result, false); // function to execute the
                prediction including "edge-impulse-sdk/classifier/ei_run_classifier.h"
            if (resp != EI_IMPULSE_OK) {
                ei_printf("ERROR: Failed to run classifier (%d)\n", resp);
                continue;
            }
            best_action = clip(result.classification[0].value, -1.0, 1.0);
            ////////////////////COMMAND//////////////////
            dutyCycle = (int)(MAX_DUTY * abs(best_action));
            mcpwm_comparator_set_compare_value(comparator, dutyCycle);
            if (best_action > 0)
                go_ccw();
             else
                go_cw();
            ///////////////STATE UPDATE////////////
            inference_end_time = esp_timer_get_time();
            dt = inference_end_time - inference_start_time;
            dt = (float)dt / 1000000;
            if (stp != 0)
            {
                th = get_angle(encoder_count);
                delta_theta = th - prev_th;
                if (delta_theta > M_PI) {
                    delta_theta -= 2.0 * M_PI;
                } else if (delta_theta < -M_PI) {
                    delta_theta += 2.0 * M_PI;
                }
                vel = delta_theta / dt;
            }
            inference_start_time = esp_timer_get_time();
            ////////////UPDATE TANK////////////////////
            u = (prev_best_action * 12 - vel * 0.07) * 0.07 / 32;
            de = u * (delta_theta);
            if (de > 0 && best_action != 0)
                tank = tank - de;
            if (tank <= 0)
                break;
            ////////////////////////////////////////////
            pos_err = normalize_angle(th);
            prev_best_action = best_action;
            prev_th = th;
            stp++;
            tot_steps++;
            printf("%d,%d,%f,%f,%f\n", ep, tot_steps, pos_err, vel, TANK_INIT - tank);
        }
    }
}
```

4.4 Inference Execution

To compare the simulation results with those obtained on the real model, the policies $\pi_{e_{inf}}$ and π_{e^*} (Table 3.1) trained earlier were exported. The performance was evaluated by observing two main metrics: the energy used to complete the task and the position error, the latter obtained by normalizing the angle θ in the interval $[-\pi, \pi]$.

During the tests, several episodes were performed, each lasting 200 steps until the tank was depleted. Between episodes, sufficient time was allowed to manually reposition the device at an angle of $\theta = 180°$ relative to the vertical. In Figure 4.9, the last five episodes are shown for brevity, but it is important to note that these episodes are preceded by other failed attempts from the policy. In the first four, the policy fails to stabilize the pendulum, continuing to oscillate in the case of policy π_{e^*} or rotating in the case of policy $\pi_{e_{inf}}$. Only in the intervals highlighted by a red rectangle, corresponding to the fifth episode, do the two policies successfully complete the control task, keeping the vertical position until the 200-step period ends.

A crucial aspect observed during policy execution on the microcontroller is the neural network inference time. Each inference cycle takes approximately 50ms, a relatively high value. This delay introduces latency in the control, making it harder to respond quickly to rapid changes in the system's state. As a result, the pendulum may become less stable, especially during phases requiring quick adjustments.

For the tank energy initialization for policy π_{e^*}, the energy limit was set to $0.4J$ based on the energy consumption observed during the execution of policy $\pi_{e_{inf}}$. This measure prevented the propagation of unstable behaviors over an extended period, as the episode was terminated once the policy exceeded the predefined energy threshold.

A more detailed analysis of the fifth episode for both policies in Figures 4.10 and 4.11 reveals that policy $\pi_{e_{inf}}$ reached equilibrium more quickly by completing full rotations, resulting in higher energy consumption. On the other hand, policy π_{e^*} oscillated around $\theta = 180°$ before stabilizing, consuming less energy but taking more time.

The results from the real model tests clearly show the differences between the two policies in terms of speed and energy consumption. However, it is important to note that both solutions have significant room for improvement, especially in terms of overall stability and the ability to respond promptly to state changes. The inference time of approximately 50ms is a key limitation: reducing this latency, along with creating a more accurate mathematical model with fewer parameter uncertainties, could substantially improve control performance, particularly when the pendulum is near the stability threshold.

Moreover, the differing behaviors between the two policies highlight the importance of a trade-off between stabilization speed and energy consumption. Policy $\pi_{e_{inf}}$, although

faster in achieving the result by applying a torque that directly brings the pendulum to the vertical, is less energy-efficient. Conversely, policy π_{e^*} favors a more conservative approach, initially acquiring energy through oscillations, consuming less overall energy but requiring more time to stabilize the pendulum. These considerations are relevant in applications where battery life or energy efficiency is crucial, or in critical contexts where human interaction demands precise and controlled distribution of mechanical energy.

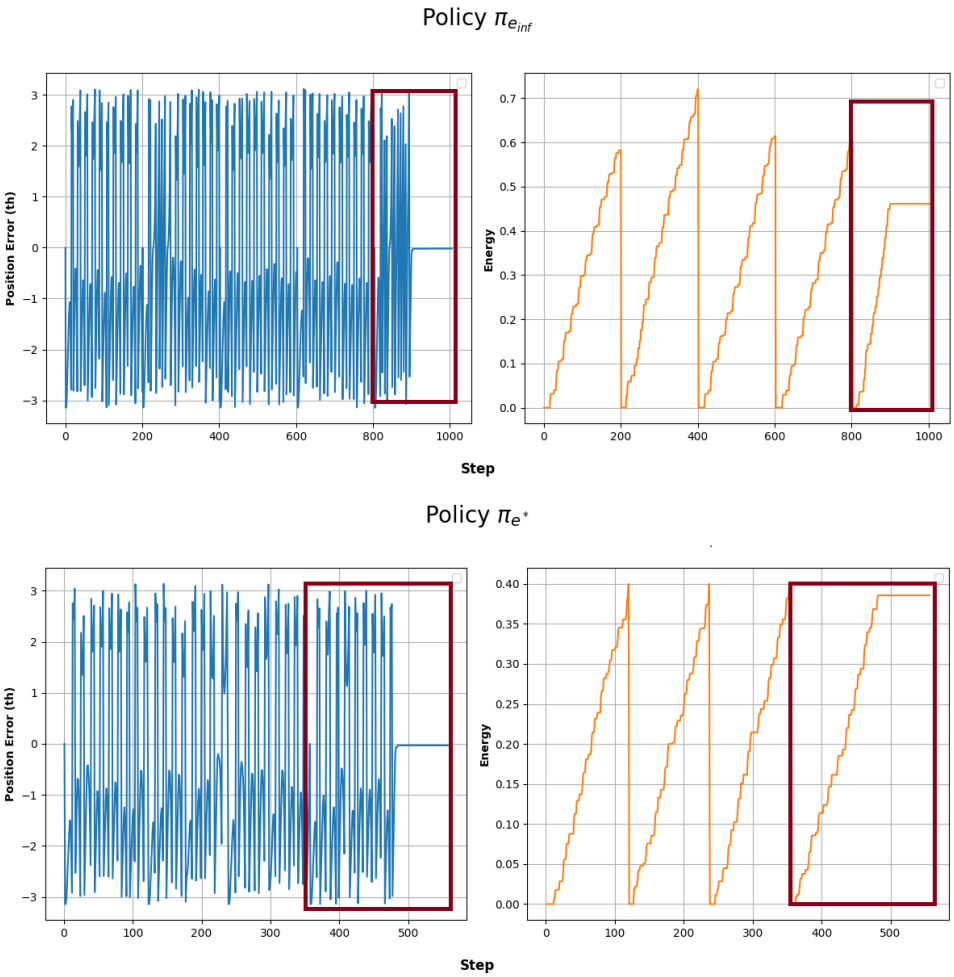


Figure 4.9: Position error on the left and energy used on the right for the five episodes considered

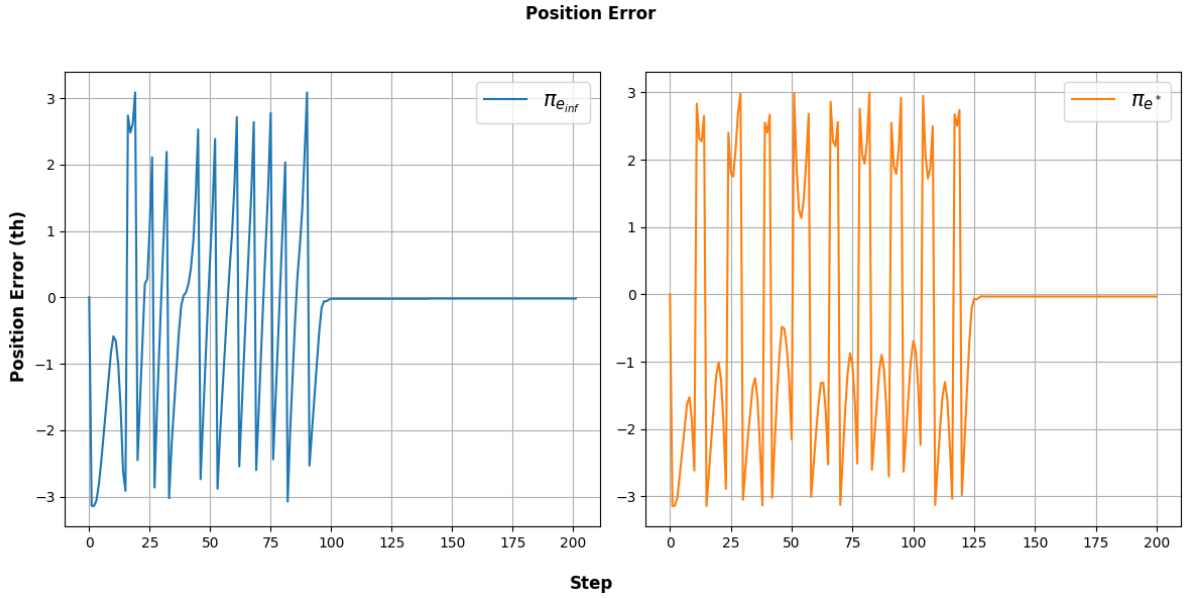


Figure 4.10

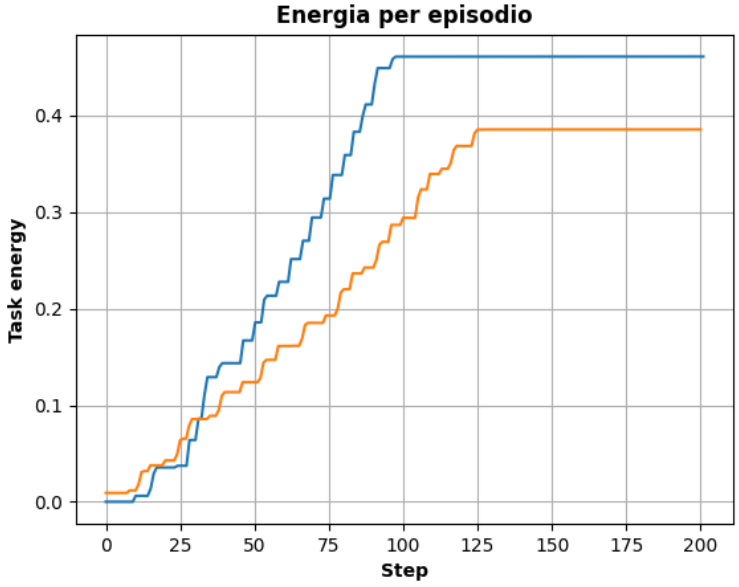


Figure 4.11

Chapter 5

Conclusions

This work explored the implementation of a passively learned control policy via reinforcement learning on a microcontroller, aiming to compare the results obtained in simulation with those of the real device. The approach was divided into multiple phases: initially, the policy was trained in simulation using a virtual energy tank to ensure system passivity. Subsequently, it was implemented on a physical system based on the ESP32 microcontroller.

The results demonstrated that the integration of reinforcement learning and energy tanks is effective not only in simulations but also on resource-constrained hardware. Tests showed that the learned policy could stabilize the pendulum with controlled energy consumption, as predicted by the trained model. Using a neural network for control allowed for inference directly on the microcontroller, maintaining good energy efficiency and a high response speed.

However, some limitations emerged during practical implementation. Notably, uncertainties such as static friction in the physical system, inaccuracies in the motor's electrical parameters, and slow neural network inference times introduced discrepancies between real and simulated results. For example, while the overall energy consumption of both policies was higher than in simulation, the system with energy constraints demonstrated greater efficiency, confirming expectations that the passive policy would consume less energy than the unconstrained one.

5.1 Future Research

Overall, this work confirms the feasibility of implementing passive policies on low-power edge devices like microcontrollers. While the results are encouraging, further experiments are needed to validate the robustness of the policies in more complex contexts or under external perturbations. This implementation can be extended and adapted for more advanced applications in industrial or robotic domains, where stability, safety, and energy efficiency are critical factors. Potential future developments include:

- Optimizing closed-loop parameters to analyze differences between real and simulated behaviors.
- Exploring new control strategies that combine the strengths of both policies, aiming for an optimal balance between speed and energy consumption.
- Achieving the desired behavior by completing training on the microcontroller to adapt the policy to the differences between simulation and reality.
- Optimizing inference to further reduce response times, downsizing models to align with the goals of Tiny Machine Learning (TinyML).

Appendix A

Current Measurement

During the development and implementation of the control circuit, a measurement chain was designed to measure the current drawn by the motor using a shunt resistor and the ESP32's ADC, enabling the calculation of the torque delivered. However, this circuit did not work as expected.
The following section describes the design of the measurement chain and highlights the main issues encountered during implementation.

INSTRUMENTATION AMPLIFIER

The voltage measured across the shunt resistor was amplified using the instrumentation amplifier *INA128PA*.

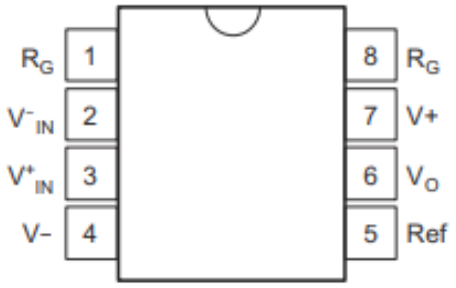


Figure A.1: Instrumentation Amplifier

The input and output signals from the component were then filtered using two identical RC filters tuned to the same frequency as the current ripple, measured via an oscilloscope, to ensure better ADC readings. The connections between the components are shown in Figure 4.7.

A.1 Measurement Chain

During the circuit design, special attention was given to the measurement chain that transmits the voltage measured across the shunt resistor (R_{shunt}) to the ADC input (V_{ADC}). The gain of the amplification stage was calculated considering the measured voltage across R_{shunt} (V_d) and the dynamic range of V_{ADC}. Based on empirical tests, the motor's current consumption varied between [0,450]mA. Thus, with $R_{shunt} = 1\Omega$, V_d ranged from [0,450]mV. However, the ESP32's SAR ADC, operating within its maximum conversion range [0,3300]mV, exhibits non-linearities. To address this, V_{ADC} was mapped to the recommended range [150,1750]mV with an input attenuation of 6dB.

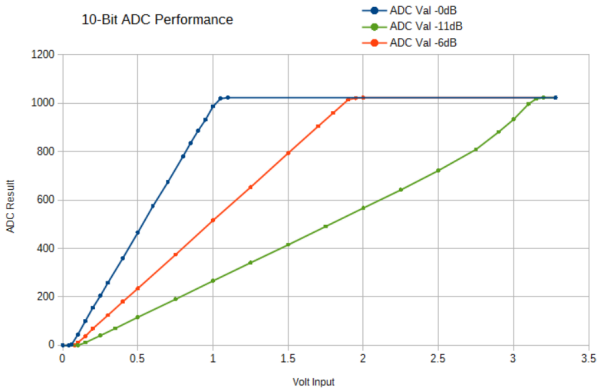


Figure A.2: ADC Non-Linearity at Different Attenuation Levels

Given the values for V_d and V_{ADC}, the amplifier gain was calculated as:

$$G = \frac{\Delta V_{ADC}}{\Delta V_d} = \frac{1600}{450} \approx 3.56$$

Thus, R_g was calculated based on the INA128 datasheet formula:

$$R_g = \frac{50K\Omega}{G-1} \approx 20K\Omega$$

Since the ADC becomes linear starting at $150mV$, an offset voltage V_{off} was added using a potentiometer for precise adjustment.

A.1.1 Challenges

Despite being fully defined, this solution was not adopted due to several reasons:

- The limited dynamic range of V_{ADC} did not allow for precise voltage measurement.
- The ADC reading task overloaded the microcontroller, occasionally causing it to lose encoder readings and introducing positional errors.

- Due to imprecise motor electrical parameters, including the torque constant, using the motor model developed in simulation proved more convenient, given the aforementioned issues.

Bibliography

[1] AIWintermuteAI. *Edge Impulse Example: standalone inferencing (Espressif ESP32)*. URL: https://github.com/edgeimpulse/example-standalone-inferencing-espressif-esp32.

[2] F. Califano et al. "On the use of energy tanks for robotic systems". In: (2022).

[3] *ESP32 WROOM32 datasheet*.

[4] Espressif. *API Reference*. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/index.html.

[5] The Farama Foundation. *Gymnasium*. URL: https://gymnasium.farama.org/index.html.

[6] Gymnasium. *Gymnasium basic usage*. URL: 'https://gymnasium.farama.org/content/basic_usage/'.

[7] Edge Impulse. *Bring Intelligence to Any Edge Device*. URL: https://edgeimpulse.com/.

[8] *LM298N Dual H-Bridge Motor Driver datasheet*.

[9] Carlos Luis. *classic control, pendulum*. URL: 'https://github.com/openai/gym/blob/master/gym/envs/classic_control/pendulum.py'.

[10] *Photoelectric Incremental Rotary Encoder 38S6G5-B-G24N*. URL: 'https://it.aliexpress.com/item/1005005071771659.html?src=google#nav-specification.

[11] Lutz Roeder. *Netron*. URL: https://netron.app/.

[12] Lasse Scherffig. "Reinforcement Learning in Motor Control". PhD thesis. University of Osnabruck, 2002.

[13] Stable-Baselines3. *Exporting models*. URL: 'https://stable-baselines3.readthedocs.io/en/master/guide/export.html.

[14] Stable-Baselines3. *Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations*. URL: https://stable-baselines3.readthedocs.io/en/master/.

[15] StableBaselines3. *RL Algorithms*. URL: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html.

[16] R. Zanella et al. "Learning passive policies with virtual energy tanks in robotics." In: *IET Control Theory Appl. 18, 541–550 (2024)* (2024).