UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

GAIT ADAPTATION OF QUADRUPED ROBOT VIA CENTRAL PATTERN GENERATOR AND REINFORCEMENT LEARNING

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

IGNACIO ANTONIO DASSORI WALKER

PROFESOR GUÍA:
MARTIN ADAMS

PROFESOR CO-GUÍA:
JORGE VÁSQUEZ ALBORNOZ

COMISIÓN:
ANDRÉS CABA RUTTE

SANTIAGO DE CHILE
2024

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO
POR: IGNACIO ANTONIO DASSORI WALKER
FECHA: 2024
PROF. GUÍA: MARTIN ADAMS

ADAPTACIÓN AL TERRENO DE ROBOT MÓVIL VÍA GENERADOR DE PATRONES CENTRALES Y APRENDIZAJE REFORZADO

Avances en la disciplina de aprendizaje reforzado profundo han mostrado promesa en la utilización de estos métodos para la locomoción de robots a través de ambientes complejos. Sin embargo, estos métodos suelen sufrir de baja eficiencia de muestras debido a espacios de acción de alta dimensionalidad. En este trabajo proponemos e implementamos una arquitectura de control para robot cuadrúpedos basada en el uso de generadores de patrones centrales y aprendizaje reforzado. Nuestro método integra extracción de características del ambiente con aprendizaje no supervisado y selección de parámetros de un generador de trajectorias cíclicas para aprender conductas de caminata adaptables a los condiciones observadas por el robot. Restrindiengo el espacio de acción de los motores de nuestro robot somos capaces de aprender caminatas simples en menos de una hora, y caminatas capaces de escalar caminos de pendiente variable en menos de tres horas. Demostramos como nuestro método supera líneas base del estado del arte, superándolas en eficiencia de muestras y rendimiento.

GAIT ADAPTATION OF QUADRUPED ROBOT VIA CENTRAL PATTERN GENERATOR AND REINFORCEMENT LEARNING

Advances in the discipline of deep reinforcement learning have shown promise in utilizing these methods for the locomotion of robots through complex environments. However, these methods often suffer from low sample efficiency due to high-dimensional action spaces. In this work, we propose and implement a control architecture for quadrupedal robots based on the use of central pattern generators and reinforcement learning. Our method integrates environment feature extraction with unsupervised learning and parameter selection from a cyclic trajectory generator to learn walking behaviors adaptable to the states observed by the robot. By constraining the action space of our robot's motors, we are able to learn simple gaits in less than an hour and gaits capable of navigating paths with varying slopes in less than three hours. We demonstrate how our method outperforms state-of-the-art baselines, surpassing them in sample efficiency and performance.

A mi madre y padre,
a quienes les debo todo.

Agradecimientos

El desarrollo de este trabajo fue un proceso largo y arduo, el cual no habría sido posible sin la presencia de ciertas personas en mi vida. Quisiera comenzar agradeciendo a mi familia, quienes me han brindado un espacio lleno de amor, comodidad y tranquilidad. A mis amigos de primer año de la universidad, con quienes compartí esta etapa de comienzo a fin llena de estudios y risas. A mis amigos del colegio, nuestras amistades tan largas como nuestras vidas. Si bien muchos no entenderán este trabajo, sepan que han tenido un profundo impacto en mi vida. A mis compañeros de eléctrica, a quienes extrañaré profundamente encontrar en los pasillos, compartir cátedras y almuerzos. Solo desearía haberlos podido conocer antes. Finalmente, a los miembros del laboratorio de visión computacional y a mis profesores, mentes brillantes con las que tuve la oportunidad de trabajar codo a codo los últimos años de mi carrera. Agradezco y aprecio a cada uno de ustedes.

Table of Contents

Table Index

Figure Index

Chapter 1

Introduction

1.1. Motivation and description of the problem

Robotics has achieved great success in the world of industrial manufacturing, where robotic arms are used every day to perform repetitive tasks at great speed and precision. However, these robots share a common problem which binds them to a particular spot: lack of mobility. In contrast, a mobile robot is one that can sense and traverse its environment autonomously [1], giving it a higher degree of flexibility to apply its talents elsewhere.

In the field of mobile robotics, legged robots show the biggest potential to traverse dynamic environments given their wide and diverse set of agile skills. However, developing a motion controller becomes challenging for high dimensional action spaces and difficult dynamic systems. A modern approach to this problem is the use of deep reinforcement learning (DRL), which learns an optimal control policy without prior knowledge of the world by interacting with its environment over many trials [2]. By defining a reward function that encapsulates the desired behaviors wanted from the robot, an optimal control strategy can be learned by maximizing the expected cumulative future reward. In particular, policy gradient reinforcement learning methods have shown success in learning complex locomotion strategies on legged robots, such as walking behaviors on bipedals [3].

Many reinforcement learning methods suffer from low sample efficiency, meaning they require large amounts of experience to learn to solve a task. In the case of legged robots this issue is especially noticeable given their many degrees of freedom, which result in a high dimensional action space. Because agents typically learn from scratch, they need to explore the whole action-state space to learn an optimal policy, which can become very sample inefficient as the complexity of the problem increases. There are many strategies that try to alleviate this issue like epsilon-greedy, batch-RL and model-based RL, among others. Low sample efficiency is also observed in tasks with long horizon where rewards are sparse. One way to solve this is through Hierarchical Reinforcement Learning, where tasks are divided into a sequence of substasks [4]. Another is to combine it with prior knowledge to guide or improve the learning process, such as intuition about the task or mathematical models that help shape the reward and actions space. Robotic locomotion in particular benefits greatly of the combination of RL with prior knowledge, as it has a substantial amount of literature that provides insights into the mechanics, dynamics, and challenges associated with the movement of robots.

In this work we explore the problem of a quadruped robot capable of adapting its gait to its current environment. In this context, the low sample efficiency problem is proposed to be addressed in two ways. First, the robot's gait can be parameterized using a Central Pattern Generator (CPG) to reduce the size of the action space to contain strictly periodic behaviors. CPGs are neural circuits found in both invertebrate and vertebrate animals that produce rhythmic motor patterns like breathing, walking and swimming without the need of a rhythmic input [5]. The system can be trained to change the CPGs parameters based on sensory information via policy gradient methods. Second, the problem can be divided into a series of subproblems by utilizing a hierarchical structure, where the high level determines the type of gait best suited to the current observations of the environment, which is then executed by a low level policy. In this work, the high-level does not correspond to a policy, but instead a pretrained Variational Autoencoder (VAE) for environment feature extraction.

Taking this into account we formulate the following hypothesis: Integrating a Central Pattern Generator and a hierarchical structure into the reinforcement learning framework for a quadruped robot adapting its gait, with the high level represented by a pretrained Variational Autoencoder, will significantly improve sample efficiency. By parameterizing the robot's gait with CPGs, the action space is constrained to strictly periodic behaviors, reducing the complexity of the learning task. Simultaneously, the VAE determines an optimal gait based on environmental observations, which is then executed by the low-level parameter selection policy. The combination of these two strategies is expected to improve the robot's learning efficiency when compared to the direct application of state-of-the-art reinforcement learning methods, and enable walking behaviors capable of traversing complex environments.

1.2. Objectives

1.2.1. General objectives

The general objective of this work is to design and train a gait adaptation controller based on reinforcement learning and central pattern generators for a simulated quadruped robot in a sample efficient manner. The agent must be able to switch between gaits depending on the sensory feedback it receives, these being in the form of an RGB image, inertial measurements and state of robot and central pattern generator. Performance of the proposed method will then validated by comparing with state of the art DRL algorithms in order to demonstrate its advantages. The quadruped robot corresponds to the Unitree Laikago robot simulated using the open-source python module PyBullet.

1.2.2. Specific objectives

- Design and train an architecture for a reinforcement learning based controller which allows a quadruped robot agent to traverse a path with varying slopes by performing online gait adaptation in a simulated environment.
- Validate the proposed algorithm by comparing with state of the art baselines in a simulated environment under similar configurations.

Chapter 2

Theoretical Framework and State of the Art

2.1. Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning that studies the mapping of state to actions so as to maximize a numerical reward signal over time by selecting the optimal actions in each situation [2]. In RL problems the learner, often called agent, is not told what the best actions are, instead it discovers them by interacting with its environment and receiving feedback in a trial and error manner. This is inspired by the idea that we humans learn certain behaviors without an explicit teacher by interacting with the world around us through our senses, acquiring an understanding of the consequences of actions and what to do to achieve goals.

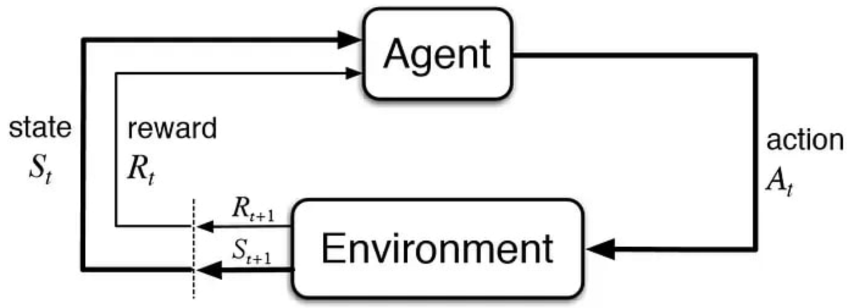


Figure 2.1: Agent-Environment interaction [2]

Figure 2.1 shows how the interaction between agent and environment in RL. Given an action, the environment returns a reward to the agent which can be thought as a feedback. This feedback can then be used to modify the way the agent chooses its actions so as to achieve greater rewards. As an example, if an infant touches the flame of a candle it will receive a negative feedback through its sensory system in the form of pain, which will influence its decision of touching it again in the future. In the following sections we will further study the elements of RL shown in Figure 2.1 and the theoretical framework on which it is based, Markov Decision Processes.

2.1.1. Markov Decision Process

Markov decision processes (MDPs) are a classical formalization of sequential decision making, where actions taken influence not only immediate rewards, but also future states and consequently future rewards too [2]. Thus taking actions involves a trade-off between immediate and delayed rewards. A reinforcement learning problem can be represented theoretically by a MDP, which will be a mathematically idealized form of the problem. An MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where:

- $\mathcal{S}$: Set of possible states called *state space.*
- $\mathcal{A}$: Set of possible actions called *action space.*
- $\mathcal{P}(s, a, s')$: State transition function, which models the environments dynamics completely. $\mathcal{P}(s, a, s') = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transitioning from state $s \in \mathcal{S}$ at time t to state $s' \in \mathcal{S}$ at time $t + 1$ after taking action $a \in \mathcal{A}$.
- $\mathcal{R}(s, a)$: Immediate reward function, gives the expected reward for taking action a in state s. Is defined by $\mathcal{R}(s, a) = \mathbb{E}[r_t | s_t = s, a_t = a]$.

In a MDP, the current state contains all relevant information that characterizes past agent-environment interactions, meaning that the probability of transitioning to state s_{t+1} and receive reward r_{t+1} at time $t + 1$ depends solely on the present state and action, not on earlier ones. This is known as the *Markov Property* and is a necessary condition for a MDP to be valid. The process follows the structure shown in Figure 2.2, where for each discrete time step the agent is in state s_t and takes an action a_t according to its policy $\pi(a_t | s_t)$. A scalar reward r_t is given by $\mathcal{R}(s, a)$ and the state transitions to s_{t+1} according to the state transition function $\mathcal{P}(s, a, s')$.

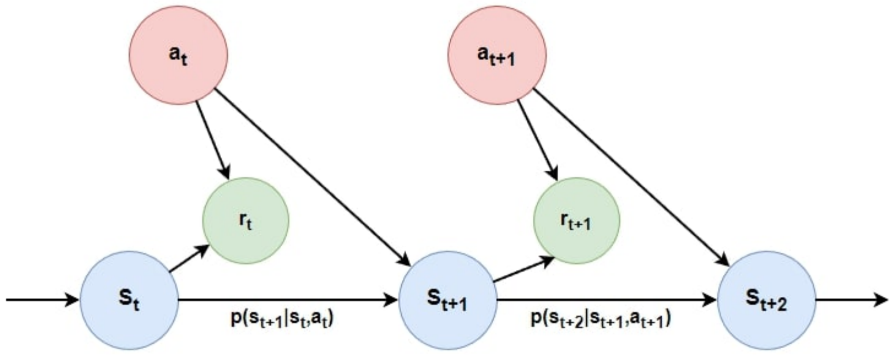


Figure 2.2: Markov Decision Process Diagram

In many cases, be it because of noisy or limited sensors, the agent cannot directly observe the underlying state of the system, so a more generalized version called *Partially Observable Markov Decision Process* (POMDP) is used in order to represent the state uncertainty. The POMDP extends the MDP model by incorporating observations and their probability of occurrence conditional on the state of the environment [6]. A POMDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O})$, where:

- $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$: Defines a MDP.
- Ω : Set of possible observations.
- $\mathcal{O}(s', a, o)$: Observation probability function, which determines the probability of observing a particular set of observations given the current state and action $\mathcal{O}(s', a, o) = \mathbb{P}(o_t = o | s_t = s', a_{t-1} = a)$. Also known as the observation model, it represents the uncertainty in state measurements.

As shown in Figure 2.3, at each time step t the agent is in state $s \in \mathcal{S}$ and receives an observation $o_t \in \Omega$ by the observation probability function $\mathcal{O}(s', a, o)$. The agent then takes action a_t according to its policy $\pi(a_t|o_t)$, taking into account that the action is conditioned by an observation of the state instead of the state itself. The reward and state transition mechanisms are the same as the ones shown for the MDP.

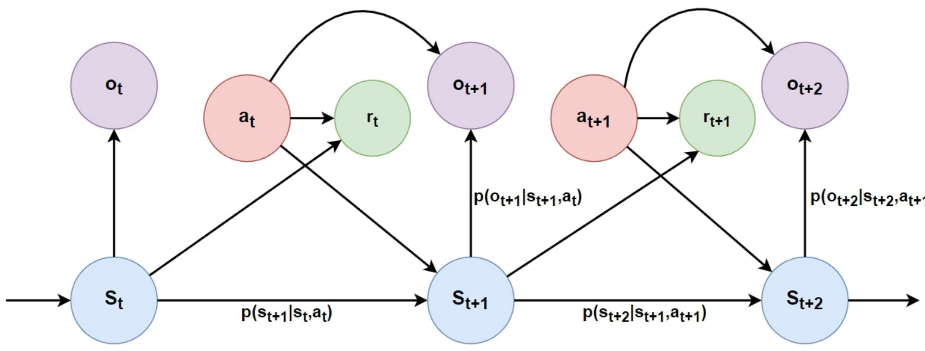


Figure 2.3: Partially Observable Markov Decision Process Diagram

2.1.2. Agent, environment, state and actions

The basic idea of reinforcement learning is that of a learning agent interacting over time with a dynamic environment to achieve a goal [2]. In the following sections we formalize the RL problem by introducing its key elements and explicit goal, as well as the fundamental Bellman equations. The core entity in the RL framework is the agent. It is a learner capable of sensing the state of the environment to a certain extent and taking actions that affect the state. The agent must have a goal, for which it optimizes its decision-making process over time by learning from its interactions with the environment. The environment is the external system with which the agent interacts. It provides feedback in the form of rewards or penalties depending on how favorable the agent's actions are to the goal.

The state represents all relevant information about the environment needed by the agent to make informed decision. States can be discrete or continuous, and the state perceived by the agent can be given by an observation model that induces uncertainty in its measurement. Actions are the mechanism by which the agent interacts with the environment and are chosen by using a policy. The action space can be discrete, where the agent chooses from a set of predefined options, or continuous, where actions are sampled from a range of values. This is typically implemented by sampling from a probability distribution.

As an example to illustrate this elements we can think of a maze solving robot. In this scenario, the robot is the learning agent and the maze is the environment. Lets say the world is discrete and we represent the maze by a grid divided into navigable and unnavigable (walls of the maze) regions. The state of the robot is its current position in the grid represented by a column and row number tuple, where the state space is limited to all navigable locations inside the maze. The actions available to the robot are also discrete, as it can choose between moving up, down, left or right. From this example it is clear to see how actions taken by the agent can affect the state of the system. If in time step t the robot's state is $s_t = (1, 1)$ and it takes actions $a_t = \text{up}$, then in time step $t + 1$ it will transition to state $s_{t+1} = (1, 2)$ assuming the state transition function $\mathcal{P}(s', a, s)$ is deterministic and $(1, 2)$ is a navigable region.

2.1.3. Rewards and policies

As discussed before, rewards are the feedback given by the environment that indicate the immediate desirability or quality of the agent's actions, and the objective of RL is to maximize the cumulative reward in the long run [2]. If the sequence of rewards received for a finite process are $r_1, r_2, \cdots, r_T$, where T is a final time step, then a return R_t can be defined as some specific function of the reward sequence. The simplest case is the sum of the rewards:

$$R_t = \sum_{t=1}^{T} r_t \tag{2.1}$$

This approach can work in applications with a clear endpoint, where there is a sequence of interactions between agent and environment that begins in an initial state and ends at time step T according to a terminal condition. This is known as an *episode* or *trial*. Going back to our example of a maze solving robot, a trip through the maze is an episode, where after reaching the terminal state (in this case the goal) the episode ends and the agent resets to the starting state. However, in may cases the agent-environment interactions are not episodic and can go on indefinitely. This are called *continuing tasks* and introduce problems with the return given by Equation 2.1, as if the final time step is $T = \infty$ then the reward we are trying to maximize could be infinite as well. To alleviate this problem we can introduce the concept of a discounted reward, where the i-th reward is discounted by multiplying it by γ^{i-1}, where $\gamma \in [0, 1]$ is a parameter called the discount rate:

$$R_t = \sum_{t=1}^{\infty} \gamma^{t-1} r_t \tag{2.2}$$

The discount rate determines the present value of future rewards. If $\gamma < 1$ the value of the infinite sum 2.2 is finite, given that the values of the reward sequence are bounded. If γ is close to 0 the agent is considered shortsighted, as it only considers short term rewards. As γ approaches 1 it becomes more farsighted, meaning it takes future rewards into account more strongly.

We have mentioned the concept of a policy being the one responsible for telling the agent which actions to take. Formally, a policy is a mapping from states to probabilities of selecting each probable action [2]. If an agent follows the policy $\pi(s|a)$ at time t, then the policy can be understood as the probability that it takes action a_t conditioned by the current state s_t. A policy can be deterministic, meaning each state has a corresponding action, or stochastic, where actions are sampled from a distribution. The job of a RL algorithm is to modify the agent's policy as a result of its experience.

The interactions between agent-environment generate a probability distribution over the sequences of pairs state-action $p_\pi(\tau) = p_\pi(s_1, a_1, \ldots, s_T, a_t)$ called trajectories. The probability of trajectory τ is given by Equation 2.3, where $p(s_1)$ is the probability of the starting state and the agent follows policy π.

$$p_\pi(\tau) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.3}$$

We can now formalize the objective of reinforcement learning mathematically. That is, to find an optimal policy that maximizes the expected value of the sum of discounted rewards received by the agent with respect to the trajectories it takes.

$$J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=1}^{T} \gamma^{t-1} \mathcal{R}(s_t, a_t) \right] \tag{2.4}$$

$$\pi^* = \arg\max_\pi J(\pi) \tag{2.5}$$

2.1.4. Value functions

To solve the issue of finding a policy that maximizes Equation 2.4 one must consider the return the agent might receive by following its current policy. For this the state-value function $V^\pi(s)$ is defined, which indicates the expected return when starting from state $s \in \mathcal{S}$ and taking actions according to policy π thereafter [2].

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=t}^{T} \gamma^{k-1} \mathcal{R}(s_k, a_k) \middle| s_t = s \right] \tag{2.6}$$

In a similar fashion, a value function for the expected value of taking action $a \in \mathcal{A}$ in state $s \in \mathbf{S}$ and then following policy π is defined. We call this the action-value function and it is given by:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=t}^{T} \gamma^{k-1} \mathcal{R}(s_k, a_k) \middle| s_t = s, a_t = a \right] \tag{2.7}$$

With value functions an ordering over policies can be defined. A policy π is said to be better than a policy π' if its expected return is greater than that of π' for all states. In other words, $\pi > \pi' \Leftrightarrow V^\pi(s) > V^{\pi'}(s),\ \forall s \in \mathcal{S}$. An optimal policy π^* is that which is better than or equal to all other policies. Then, we define the optimal state-value and action-value functions as follows:

$$V^*(s) = \max_\pi V^\pi(s) = V^{\pi^*}(s) \tag{2.8}$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) = Q^{\pi^*}(s, a) \tag{2.9}$$

2.1.5. Bellman equations

A fundamental property of the value functions $V^\pi(s)$ and $Q^\pi(s, a)$ is that they can be written in terms of each other. For the state-value function the only input is the state s, and thereafter all actions are decided by the policy. This is equivalent to taking the expected value of $Q^\pi(s,a)$ given that action a is distributed by the policy $\pi(s|a)$. For the action-value function we can take the first reward $\mathcal{R}(s,a)$ out of the sum. Given s and a, we know the next state s' is distributed according to the state transition function $p(s'|s,a)$, and therefore the rest of the sum is equivalent to taking the expected value of $V^\pi(s')$ multiplied by the discount factor.

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)} Q^\pi(s,a) \tag{2.10}$$

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)} V^\pi(s') \tag{2.11}$$

From these equations a recurrent relationship can be established for both functions, known as the Bellman expectation equations. These equations express the relationship between the value function of the current state and the value function of the next state [7].

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}[\mathcal{R}(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)} V^\pi(s')] \tag{2.12}$$

$$Q^\pi(s,a) = \mathcal{R}(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)} \mathbb{E}_{a\sim\pi(a|s)} Q^\pi(s,a) \tag{2.13}$$

Bellman equations are essential for RL as they enable the agent to estimate the values of states and actions considering future interactions. By iteratively solving these equations an algorithm can converge to an optimal value function and policy, as it the case in dynamic programming and TD methods such as SARSA and classical Q-Learning [7]. Because V^* and Q^* defined earlier are value functions for a policy, they must satisfy the self-consistency conditioned given by the Bellman equation. However, their Bellman equations can be written differently without reference to any specific policy, given that for an optimal policy we can say that the action taken will be the one which maximizes the return. These are known as the Bellman optimality equations:

$$V^*(s) = \max_{a\in\mathcal{A}} \left[\mathcal{R}(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)} V^*(s')\right] \tag{2.14}$$

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \max_{a\in\mathcal{A}} \mathbb{E}_{s'\sim p(s'|s,a)} Q^*(s,a) \tag{2.15}$$

2.1.6. Exploring vs Exploiting

When training an agent using reinforcement learning a dilemma comes up: Should the agent select actions based on which ones yield the highest expected reward? Or should it choose a different option in order to explore unknown territory? This trade-off between exploiting current knowledge and exploring the unknown is fundamental when trying to maximize long-term rewards. On one hand a greedy agent who always chooses the action with the highest immediate reward can potentially converge to a local optimum, missing out on better actions or undiscovered parts of the state space. On the other hand, an agent who explores excessively can lead to an inefficient training that focuses too much on acquiring knowledge and not using it to solve the problem.

Exploitation, as the name suggests, exploits the current knowledge of the actor by selecting the actions that are expected to yield the highest immediate reward. This kind of behavior is ill suited for an agent with poor understanding of the environment. Conversely, an agent with more experience might benefit from this approach as it refines the trajectories it knows give a high reward.

Exploration involves taking actions that are not necessarily the optimal ones given the current knowledge of the agent. This can result in finding previously unknown strategies that can, in the long run, give a much higher reward than that obtained by following the current policy. However, as stated before, too much exploration can make the algorithms inefficient.

A method that balances both approaches is the $\epsilon-greedy$ method. A parameter ϵ indicates the probability that the agent will choose an action randomly, otherwise it will choose the action which maximizes the immediate expected reward. A good way to implement this is to also make the value dynamic, meaning that at the start of training ϵ will be closer to 1, meaning the agent will explore most of the time. As episodes pass or as the performance increases, the value of ϵ decreases. As ϵ gets closer to 0 the agent chooses to exploit its knowledge and go for the greedier options.

2.2. Reinforcement Learning Taxonomy

The family of RL algorithms encompasses a vast amount of methods that aim to solve the sequential decision-making process. How the different methods approach this problem varies greatly, so to get a better idea of the types of RL algorithms that exist we introduce the main branching points that highlight the most foundational design choices in RL. A taxonomy of algorithms in modern RL is shown in Figure 2.4. The taxonomy shown is by no means all-encompassing as many algorithms mix and match properties and does not include more advanced material such as transfer learning, meta learning and exploration, as stated by the creator of the sketch [8].

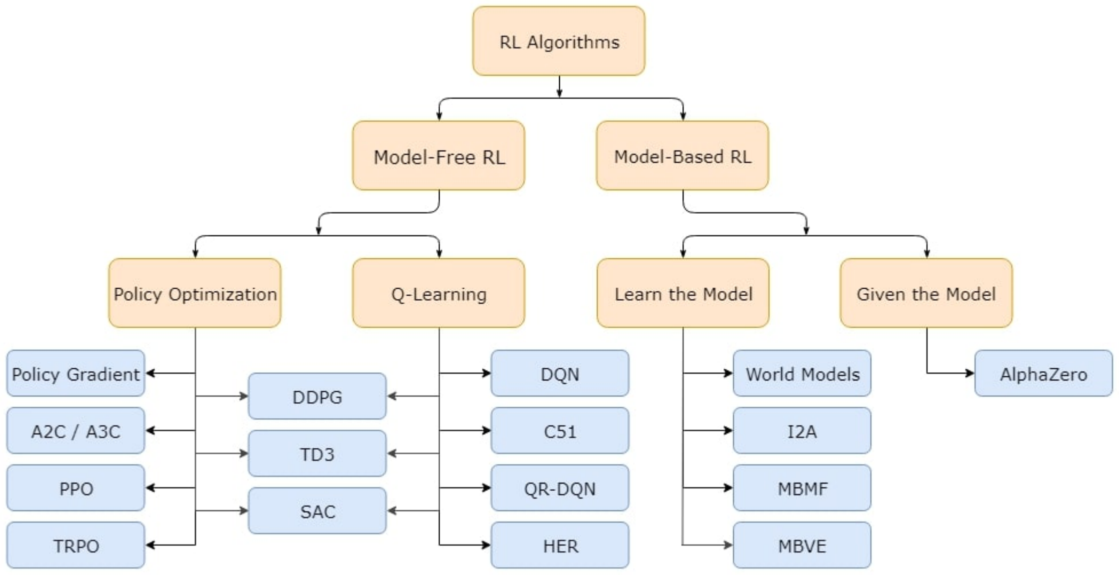


Figure 2.4: Taxonomy of algorithms in modern RL [8]

When talking about types of RL algorithms, one of the main branching points comes in whether or not the agent has access to a model of the environment. In a more practical sense, this means it has knowledge about the environments dynamics and uses it to make decisions and improve its policy. RL algorithms are divided into Model-Free and Model-Based.

2.2.1. Model-Free

Model-Free algorithms, as the name implies, have no access to any type of model of the environment. One key branching point in these methods lies in what the agent learns. The usual elements to be learned are policies, action-value (Q-functions) and state-value functions [8]. There are two main approaches in modern model-free RL called Policy Optimization and Q-Learning.

Policy Optimization: This family of methods utilizes a parameterized policy $\pi_\theta(a|s)$, where the parameters θ are optimized to maximize the expected cumulative rewards. This is typically achieved by representing the policy as a neural network with learnable weights and performing gradient ascent on the performance objective $J(\pi_\theta)$. Usually an approximation V_ϕ of the value function is also learned, which is used for estimating improvements on the policy through the advantage function (further discussed in section 2.3). The algorithms in this family vary depending on how they implement the policy optimization. For example, the A3C algorithm [9] performs gradient ascent to directly maximize performance in an asynchronous manner, executing multiple agents in parallel, on multiple instances of the environment. Another method, Proximal Policy Optimization [10], maximizes its performance indirectly by optimizing a “surrogate” objective function.

Q-Learning: These methods learn the optimal policy implicitly by learning parameterized approximations $Q_\theta(s, a)$ of the optimal action-value function $Q^*(s, a)$. The update on this methods is usually achieved by using the previously shown Bellman expectation and optimality equations. When we refer to implicitly learning the policy, this is because the action taken will be the one that maximizes the value of Q_θ, as shown in Equation 2.16. The most well known example of a Q-Learning method is the one responsible for kicking off the field of deep RL, DQN [11]. This method approximates the action-value function via a convolutional neural network, successfully implementing a RL controller for a high-dimensional sensory input.

$$a(s) = \arg\max_a Q_\theta(s, a) \tag{2.16}$$

Actor-Critic: Policy optimization and Q-Learning, while different in how they approach the RL problem, are not incompatible. Many algorithms apply both methods in an actor-critic framework, where π_θ is the actor responsible for taking actions and Q_ϕ is the critic who estimates the value and provides an evaluation of the current policy. Some popular actor-critic algorithms are DDPG [12], which learns both a deterministic policy and a Q-function by using them to improve each-other, and SAC [13], a variant that utilizes stochastic policies and entropy regularization for a more stable learning.

2.2.2. Model-Based

Since this work focuses on the application of model-free methods, we will not be going over these methods extensively. As mentioned before, model-based methods give the agent access to a model of the environment. These methods are separated into to categories: Those in which the model is given and those in which it is learned. One basic approach is to not explicitly represent the policy, but instead use model-predictive-control to select actions. Some algorithms like MBMF explore the learned environment using MPC [8], learning and updating its knowledge of the environments dynamics. Other methods rely on an expert model, where an explicit policy is trained by comparing its actions with those taken by the expert. While these methods are usually more sample efficient than model-free ones, they often require expert knowledge of the problem.

2.2.3. Types of learning loops

The training loop of reinforcement learning usually involves a rollout phase, where data is generated by the interaction of the agent with the environment under policy π, and an update phase where the collected data is used to optimize the policy [14]. There are three main learning methods used in RL: Online, off-policy and offline reinforcement learning. In classical online RL the policy π_k is updated with rollout data collected by π_k itself. After the update the cycle continues using the updated π_{k+1} policy to collect data. When using off-policy RL, the experiences of the agent are stored in a data buffer (also called replay buffer) $\mathcal{D}$. The name off-policy comes from the fact that the agent's policy used for selecting actions is different from the policy being improved. This is because the experiences used for training a new policy π_{k+1} are sampled from the replay buffer $\mathcal{D}$, which presents experiences generated by all previous policies $\pi_0, \dots, \pi_k$. A more recent concept which follows the trend of scalable data-driven machine learning methods is offline RL. It employs a static dataset $\mathcal{D}$ generated by some (potentially unknown) policy π_β, from where experiences are sampled to fully train a policy π. In contrast with the aforementioned methods, data is collected only once before the training even begins, and the training process does not interact with the creation of new rollouts, meaning the policy is only deployed after being fully trained. This type of learning aims to take advantage of large amounts of previously collected offline data. However, this approach has its issues, as a dataset of experiences might not (and most like will not) cover the entire state-action space, which might lead to poor generalization and difficulties in handling unseen states and actions.

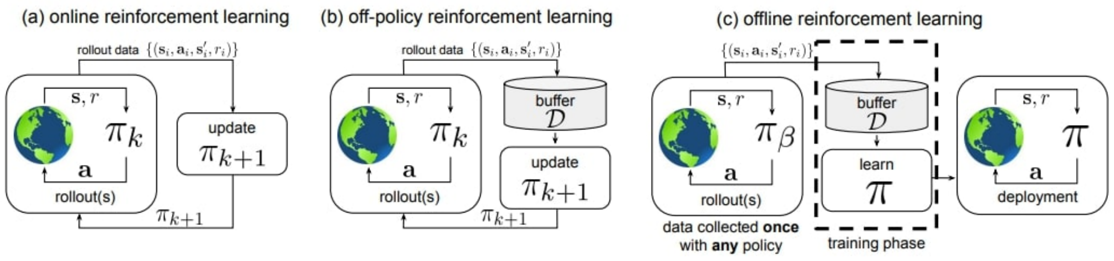

(a) online reinforcement learning
rollout data $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
$\mathbf{s}, r$
π_k
$\mathbf{a}$
rollout(s)
update
π_{k+1}
π_{k+1}
(b) off-policy reinforcement learning
rollout data $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
$\mathbf{s}, r$
π_k
$\mathbf{a}$
rollout(s)
buffer $\mathcal{D}$
update
π_{k+1}
π_{k+1}
(c) offline reinforcement learning
$\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
$\mathbf{s}, r$
π_β
$\mathbf{a}$
rollout(s)
buffer $\mathcal{D}$
learn
π
$\mathbf{s}, r$
π
$\mathbf{a}$
deployment
data collected **once** with **any** policy
training phase

Figure 2.5: Types of learning loops in reinforcement learning. (a) Online, (b) off-policy and (c) offline learning [14].

2.3. Deep Reinforcement Learning

Reinforcement learning is by no means a modern technique, as its earliest foundations can be attributed to Richard Bellman all the way back in the 1950s with the development of dynamic programming [15]. While classical RL is certainly useful it lacks scalability and is therefore limited to fairly low-dimensional problems [16]. However the paradigm of artificial intelligence began to change with the rise of deep learning, where deep neural networks began being used to approximate complex functions in the fields of supervised learning. It wasn't long until these methods began being applied to reinforcement learning in order to solve previously intractable problems, such as teaching an agent to play Atari games at human level [11] or defeating the Go world champion [17].

In supervised learning, input data is fed into a neural network in a process called forward propagation, where computations are performed layer by layer. The output of the final layer is the prediction, which can be compared with a ground truth (usually the true label of the input) through a loss function to compute its error. Computing the error's gradient with respect to the weights of the networks allows us to update the weights in the direction that minimizes loss. The loss can be propagated backwards through the network, and computing the gradient for each layer helps us optimize the model by using gradient descent. Even though reinforcement learning does not have access to labeled data, we will see that a pgradient can be computed.

2.3.1. Simplest Policy Gradient

Lets consider the case of a stochastic, parameterized policy π_θ. The goal of RL is to find a policy that maximizes $J(\pi_\theta)$, so we would like to optimize the policy by gradient ascent, where as in supervised learning we would use descent to minimize a loss.

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2.17}$$

The gradient of the expected return is called the policy gradient, and algorithms that optimize the policy this way are called policy gradient algorithms [8]. We show the mathematical derivation of the simplest policy gradient:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)} \left[\sum_{t=1}^{T} \gamma^{t-1} \mathcal{R}(s_t, a_t) \right] \\
&= \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)} \left[R(\tau) \right] \\
&= \int \nabla_\theta p_{\pi_\theta}(\tau) R(\tau) \mathrm{d}\tau \\
&= \int p_{\pi_\theta}(\tau) \nabla_\theta \log(p_{\pi_\theta}(\tau)) R(\tau) \mathrm{d}\tau \\
&= \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)} \left[\nabla_\theta \log(p_{\pi_\theta}(\tau)) R(\tau) \right] \\
&= \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \\
&= \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} \gamma^{t-1} \mathcal{R}(s_t, a_t) \right) \right]
\end{aligned} \tag{2.18}$$

The resulting policy gradient is an expectation, meaning we can estimate it by sampling N trajectories and calculating the mean value:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{N}\sum_{k=1}^{N}\left[\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(a_t^{(k)}|s_t^{(k)}\right)\left(\sum_{t=1}^{T}\gamma^{t-1}\mathcal{R}\left(s_t^{(k)}, a_t^{(k)}\right)\right)\right] \tag{2.19}$$

2.3.2. Reward-to-go

Examining the expression for the policy gradient in Equation 2.18 we notice something which doesn't make much sense: The log-probabilities of each action are multiplied by $\mathcal{R}(\tau)$, that is, the sum of all rewards obtained in trajectory τ. This means that for the pair state-action at time t we are taking into consideration rewards obtained in the past, which contradicts the notion that the agent should maximize cumulative future rewards. In other words, the agents actions should only ever by evaluated on the basis of their consequences. The policy gradient can be expressed in a different form, called the reward-to-go policy gradient. In this form we don't consider rewards from time steps $t' < t$:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim p_{\pi_\theta}(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t'=t}^{T}\gamma^{t'-t}\mathcal{R}(s_t, a_t)\right)\right] \tag{2.20}$$

2.3.3. Baselines and Advantage Function

Simple policy gradient methods often suffer from high variance in their estimated gradients, which can negatively impact the learning stability. As we saw before, one of the terms in our policy gradient equation is the return or sum of discounted future rewards. Due to randomness in the environment and the agent's actions, these term can have a high variance, and therefore the gradient as well. Baselines are introduced to provide a reference point that allows the gradient estimates to be centered around a certain value, reducing the variance of the estimates. This can be achieved by subtracting a baseline b from the estimated return:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim p_{\pi_\theta}(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t=1}^{T}\gamma^{t-1}\mathcal{R}(s_t, a_t) - b\right)\right] \tag{2.21}$$

The use of baselines is mathematically valid, as it can be proven that it does not introduce any bias in the gradient estimate [2]. The most common choice of baseline is the on-policy state-value function $V^\pi(s_t)$, which if we recall is the expected return given state s_t. Another valid form used in many policy optimization methods [9], [10], replaces the discounted future reward term with the action-value function $Q^\pi(s_t, a_t)$ and uses the baseline we just mentioned:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{N}\sum_{k=1}^{N}\left[\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(a_t^{(k)}|s_t^{(k)}\right)\underbrace{\left(Q^\pi\left(s_t^{(k)}, a_t^{(k)}\right) - V^\pi\left(s_t^{(k)}\right)\right)}_{A^\pi\left(s_t^{(k)}, a_t^{(k)}\right)}\right] \tag{2.22}$$

Where A^π is called the advantage function and is a measure of the quality of an action give a certain state, and N is the number of sampled trajectories on which the gradient is being computed.

2.3.4. Proximal Policy Optimization

The primary deep reinforcement learning algorithm utilized for the training of policies in this work is PPO. This method aimed to address some of the issues present in the algorithms that came before it. One may think it efficient to use a sampled trajectory multiple times to perform updates on the policy, however in traditional policy gradient methods this has empirically shown to lead to destructively large updates [10], causing the policy to diverge. Sampled trajectories can only be employed once and are then discarded, leading to poor sample efficiency. Trust region methods limit the size of policy updates within an acceptable range, helping to mitigate issues such as policy oscillation and divergence. In TRPO [18], a "surrogate" objective function is maximized subject to a constraint which limits the KL-divergence between old and new policies.

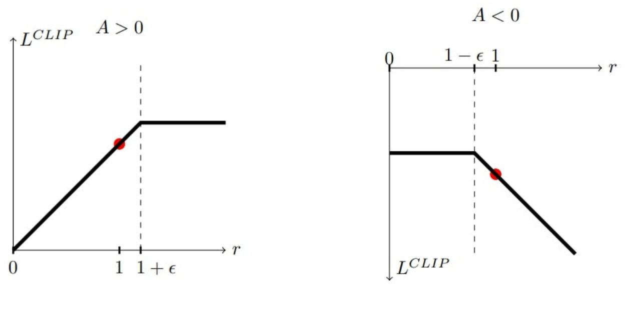
$$\underset{\theta}{\text{maximizes}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] \tag{2.23}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \tag{2.24}$$

Implementing TRPO can be intricate since it involves the utilization of complex algorithms and the need to perform both linear and quadratic approximations. PPO proposes a much simpler surrogate objective which accomplishes the same goal of constraining policy updates to a trust region by using a clipped objective. Let $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ denote the probability ratio between new and old policies,

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{2.25}$$

where ϵ is an hyperparameter that indicates how much r_t can move away from 1 before we start penalizing. The first term is the unclipped objective as seen in TRPO, while the second applies a clipping to the probability ratio, removing the incentive to move outside the interval $[1-\epsilon, 1+\epsilon]$. The min operator ensures that a lower bound of the unclipped objective is always taken. If the advantage is positive, we want to reinforce the change in that direction but limiting the size of the update, so as to not destroy our policy based on one estimate. The same applies for a negative advantage where $r_t < 1-\epsilon$, where we want to make those actions even more unlikely but we limit the update. The only case in which an unbound updated is used is when the advantage is negative, and the last update made the unfavorable actions more likely.



.

Figure 2.6: Plot showing the clipping of the surrogate objective L^{CLIP} for positive and negative advantages as a function of the probability ratio [10]

2.4. Hierarchical Reinforcement Learning

As we've previously stated, in RL an agent explores the state and action spaces by executing sequences of state-actions. For each problem the average length of these sequences is called the task horizon. For long horizon tasks, the performance of standard RL methods becomes poor as the state and actions spaces get larger. Hierarchical reinforcement learning (HRL) decomposes such long-horizon learning tasks into a hierarchy of subtasks to solve [4]. Higher-level policies select the optimal subtask, which is then resolved by a lower-level policy, effectively reducing the long horizon task into a sequence of short horizon ones. Using this structure higher level tasks are usually more abstract, and the actions taken persist for a prolonged period of time. During this time, the low level policy performs a sequence of more concrete actions until the subtask is completed or terminated. As an example, let's think about a humanoid agent whose goal is exit a closed room, which we can divide in a sequence of substasks. A high level policy makes decisions at a more abstract level, selecting actions like "walk forward", "turn left", "turn right" and "open door". After selecting a subtask, a low level policy executes finer actions required to achieve the subtask. These actions could be the movement of the body's joints, which in sequence could result in a more abstract behavior, like walking.

HRL algorithms are promising for long horizon tasks, as they have shown to outperform standard RL in problems such as continuous control, long-horizon games and robot manipulation [4]. One of the earliest formulations of HRL corresponds to the *options framework* proposed by Sutton et al. in 1999 [19]. Options are temporally extended actions that can be executed repeatedly until a certain condition or termination criterion is met. They allow the agent to operate at a more abstract level and simplify the decision-making process. In this framework options and their termination conditions are predefined. More modern approaches make use of policy gradient methods to automatically discover subtasks [20] and learn termination conditions [21].

An advantage of separating problems into subtasks is that it enables the reuse of low level policies for transfer learning. Low level policies often learn primitives, fundamental actions that an agent can take in an environment. If we are using the same agent to solve a different problem which uses similar fundamental actions, there is no need to start from scratch. Let's return to our previous example of a humanoid agent, but this time the goal is to follow a path up a hill. While the problem to solve is different, the fundamental actions are very similar, like the ability to walk and turn. Of course, walking up a slope is not the exact same as walking on flat terrain, but they are relatively close in the action space, making it more efficient than starting from scratch. In [22] the authors trained a quadruped robot for a goal finding task, and then successfully transferred the low level policy to solve a cliff traversal task. The results showed the agent learned the desired behaviors faster than if it had started from scratch.

2.5. Locomotion for Legged Robots

Legged robot locomotion refers to the act of designing and controlling robotic systems capable of walking, running, or moving using legs as the primary means of propulsion. Inspired by the diverse locomotion abilities of animals, legged robots aim to replicate or surpass their agility and adaptability across various terrains. Legged locomotion presents unique challenges in robotic engineering, involving the coordination of multiple limbs, balance control, terrain adaptation, and efficient energy usage. By employing advanced control algorithms, mechanical designs, and sensor integration, legged robots strive to achieve versatile locomotion, enabling them to navigate complex environments, traverse uneven surfaces, and perform tasks that are difficult or inaccessible for wheeled or static robots. In this work we focus on two main research areas in legged locomotion: The use of Central Pattern Generators and RL.

2.5.1. Central Pattern Generators

CPGs are neural circuits found in both invertebrate and vertebrate animals than produce rhythmic motor patterns like breathing, walking and swimming without the need of a rhythmic input [5]. Inspired by this biological phenomenon, CPGs have been used in open-loop gait generation to great success [23]. More recent works have implemented close-loop CPG by integrating some type of sensory feedback to adapt to adapt to challenging terrain [24, 25]. CPGs are usually expressed as a set of coupled oscillators, which can be modeled as a system of coupled ordinary differential equations. For a n-legged robot, let $x(t) = [x_1, ..., x_n]$ and $y(t) = [y_1, ..., y_n]$ be the angles of the shoulder joints of each leg in the axial and sagital plane respectively. A basic CPG model with a circular limit cycle is described by the following equations:

$$\begin{aligned} \dot{x_i}(t) &= -\omega \cdot y_i(t) + \gamma \left(\mu^2 - \sqrt{x_i(t)^2 + y_i(t)^2} \right) \cdot x_i(t) \\ \dot{y_i}(t) &= +\omega \cdot x_i(t) + \gamma \left(\mu^2 - \sqrt{x_i(t)^2 + y_i(t)^2} \right) \cdot y_i(t) + (\lambda \Sigma_j K_{ij} y_j(t)) \end{aligned} \tag{2.26}$$

Where μ is the radius of the limit cycle, which defines a closed path for the joint to follow, ω is the angular frequency and γ is the forcing to the limit cycle. The extra term in the second equation defines the gait of the robot by setting a phase relationship between the legs. We can note that in a RL application this term can be ignored, as gaits are learned during training. A more advanced formulation of a CPG that considers sensory feedback can be found in [25].

2.5.2. RL approaches

The use of RL to learn agile locomotion skills for legged robots has been studied mostly on simulated environments given that poor sample efficiency makes it really hard to train for extensive periods of time on real hardware. Many of the works tackle the problem of learning to adapt to complex environments or completing a task. In [26] the authors used deep reinforcement learning in a two-level hierarchy, comprising a high-level gait planner and a low-level gait controller. They tested their architecture on a simulated quadruped platform, achieving great success in a suit of challenging terrains. The authors of [22] included visual inputs as sensory feedback, separating the problem hierarchically in vision and motor control modules. The modules were represented by neural network policies and trained using evolutionary strategies. Some works propose hierarchical frameworks that combine model-free RL with classical mode-based control. [27] proposes a control framework that combines Model Predictive Control with RL, where the high level is a RL policy responsible for selecting both the optimal gait and model parameters for the low-level MPC. As such, the low level does not learn a control policy, greatly reducing the action-state search space. However, such approaches require exact knowledge of the dynamic model. Finally, some works that parameterize their primitives as cyclic movements [28, 29] where able to validate their results on real hardware with moderate training times.

2.6. Variational Autoencoders

Autoencoders are a type artificial neural networks used for learning feature representation of high dimensionality inputs, often used in unsupervised learning to obtain codings of unlabeled data. It consists of two networks: An encoder that compresses the input, mapping it to a point in the latent space; and a decoder, which takes this latent point and from it tries to reconstruct the input. The procedure effectively reduces the dimension of data by learning to extract relevant features, something quite useful in the field of machine learning which often suffers from the curse of dimensionality. The loss functions used for training are also quite intuitive, as they should represent a reconstruction error. Depending on the type of data, MSE or Binary Cross Entropy can be used to compare input and output.

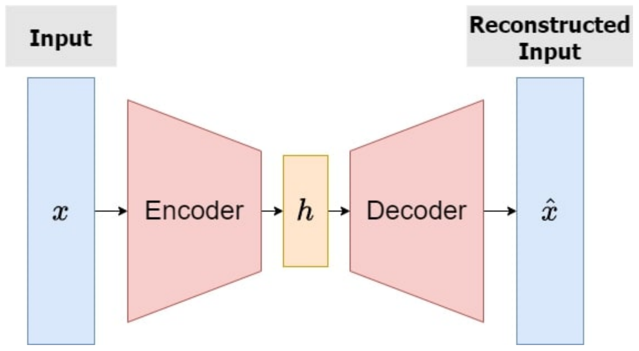


Figure 2.7: Architecture of traditional autoencoder.

A Variational Autoencoder (VAE) represents a modified version of the traditional autoencoder architecture, incorporating probabilistic and generative elements. It is often considered an upgrade due to its capacity for learning representations of complex data distributions and enhancing the generalization of the latent space [30].Unlike the traditional autoencoder, where the encoder learns to map inputs to specific points in the latent space, a VAE's encoder is designed to map inputs to probability distributions over the latent space. For instance, it may output parameters of a normal distribution (μ,σ^2) that represent the statistical properties of the latent space. Building on the probabilistic nature of the encoder, a latent vector z is then sampled from the learned distribution, utilizing a technique known as the reparameterization trick. This stochastic sampling gives the network its generative capabilities, as outputs are no longer deterministic. The decoder, for the most part, remains the same, learning to reconstruct the output from the latent vector. The introduction of sampling from the latent space enables the VAE to generate diverse and continuous outputs during both training and inference. The revised architecture can be seen in Figure 2.8. It is relevant to note that, while we show the output of the decoder as mean and standard deviation, any parameterized distribution can be employed in the latent space, providing a high degree of flexibility. This flexibility is particularly advantageous when there is prior knowledge about the data distribution.

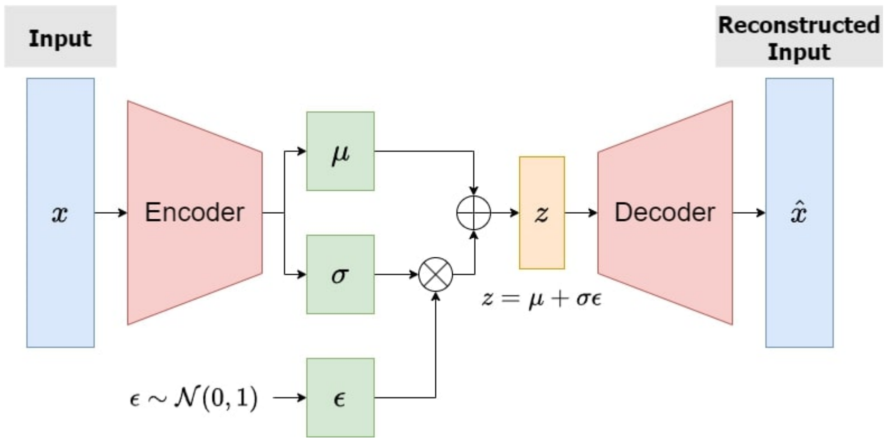


Figure 2.8: Architecture of variational autoencoder

The sampling process introduces a challenge, as directly sampling from $\mathcal{N}(\mu, \sigma^2)$ creates a non-differentiable operation within the network, making it impossible to backpropagate a gradient. As previously mentioned, we employ the reparameterization trick, a crucial component illustrated in our architecture diagram. In essence, the latent vector is sampled using the formula $z = \mu + \sigma\epsilon$, where ϵ is an auxiliary noise variable $\epsilon \sim \mathcal{N}(0, 1)$. This formulation effectively separates the stochastic element from the deterministic parameters μ and σ, allowing for the computation of gradients. Since there is no need to backpropagate through the auxiliary noise term it is of no consequence that it is randomly sampled.

While we have the ability to propagate gradients throughout the network, a crucial element is the inclusion of a loss term. While a straightforward reconstruction error could serve this purpose, its use alone falls short in guiding the model to discover meaningful distributions. Relying solely on reconstruction error might lead to an irregular latent space with limited generative capabilities. To address this, during training the objective is to minimize the loss function depicted in Equation 2.27, which consist of two terms. The first is called the regularization term, where D_{KL} represents the Kullback-Leiber Divergence, $q_\phi(z|x)$ is the learned distribution of the latent variable z given by the encoder network with parameters ϕ, and $p_\theta(z)$ is a prior for z. This term measures how much the learned distribution diverges from the prior and penalizes it, encouraging the latent space to follow a certain structure. A standard normal distribution is often used for $p_\theta(z)$. The second term is a reconstruction loss, where $q_\theta(z|x)$ is the likelihood of reconstructing the input data x given a latent sample z, decoded by the network with parameters θ.

$$loss = -D_{KL}(q_\phi(z|x)||p_\theta(z)) + \mathbb{E}_{q_\phi(z|x)}[\log p(x|z)] \tag{2.27}$$

As mentioned for autoencoder, the reconstruction loss can be calculated by comparing input and output with some distance metric, like MSE and Binary Cross Entropy. As for the regularization term, in the Gaussian case it can be computed quite easily as shown in [30]. The term can be expressed as:

$$D_{KL}(q_\phi(z|x)||p_\theta(z)) = \frac{1}{2}\sum_{j=1}^{J}(1 + \log\left(\sigma_j^2\right) - \mu_j^2 - \sigma_j^2) \tag{2.28}$$

Chapter 3

Methodology

In this section we describe the methodology followed during the development of this work. First, we give a brief description of the necessary tools and how they will be utilized. We then present the proposed architecture and the step-by-step process employed for the development of the control system. All the code written for this work can be found in the project's Github repository.

3.1. Tools

3.1.1. PyBullet

PyBullet is a versatile and powerful physics engine and robotics simulation library built as an open-source extension of the Bullet Physics Engine [31]. PyBullet offers a Python interface and a detailed quickstart guide, making it accessible and easy to use for seasoned Python users. With this engine, users can simulate and study complex articulated robots with accurate and realistic simulations of real world physics. PyBullet integrates seamlessly with popular machine learning frameworks such as reinforcement learning, this being the reason it will be used for developing and training our RL agent in a virtual environment.

3.1.2. OpenAI Gym

Gym is an open-source Python library for developing and evaluating reinforcement learning algorithms [32]. It provides a standard API for communicating between environments and RL algorithms, as well as a vast collection of pre-defined environments that cover a diverse range of problems, such as classic control tasks and robotic manipulation. Gym supports the creation of custom environments, which will be used in this work for training on a self made, randomly generated path for the robot to traverse. Moreover, PyBullet can easily be used with Gym environments.

3.1.3. Pytorch

PyTorch is one of the main libraries used for deep learning implementations [33]. Developed by *Facebook*, it enables GPU accelerated tensor computation and automatic gradient computation for deep neural networks. This will be the chosen framework over alternatives like TensorFlow because of its accessibility and personal familiarity.

3.1.4. Stable Baselines

Stable Baselines3 (SB3) is a collection of high quality and reliable implementations in PyTorch of popular reinforcement learning algorithms developed by OpenAI [34]. The library is built on top of OpenAI Gym, which allows the training of models using a standardized framework of environments, including custom ones. The policies learned in this work were trained using the implementations found in SB3.

3.1.5. Laikago Quadruped Robot

This work focuses on the control of a quadruped robot, and therefore it is essential to have access to an open-source URDF (Unified Robot Description Format) of one. Thankfully PyBullet includes a fully articulated model of a Unitree Laikago robot. The Laikago is a quadruped with 12 degrees of freedom, represented by 3 motors in each of its legs. The motors positions can be commanded to a new position, where an internal PID controller performs the motion. The engine keeps track of the robot position and orientation which can be accessed at any time, acting as our sensory information about the state of the robot.

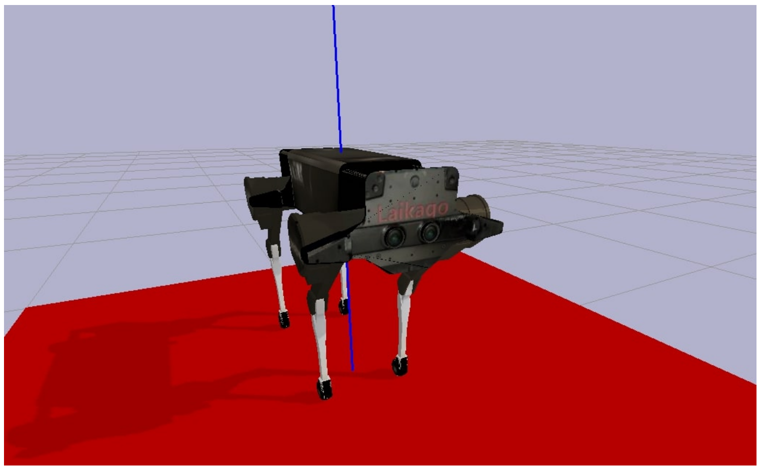


Figure 3.1: Simulated Laikago quadruped robot

Additionally, the PyBullet interface allows for the use of a debug camera, which broadcasts RGB, depth, and segmented images. Using the orientation of the robot and rotation matrices, the camera was situated from the point of view of the robot facing forward.

3.2. Proposed architecture

The complete structure of the proposed controller is presented in Figure 3.2, which is divided into three main modules: A VAE that receives RGB images from a camera positioned at the front of the robot, and transforms them into low dimensionality latent representations of the terrain; a Fully Connected Policy Network that chooses the parameter values of a central pattern generator and modulates its output by adding correction terms to the desired position of every motor; and the CPG, a mathematical representation of a cyclical gait whose trajectory is determined by a set of parameters. This way, the policy adapts the robots gait to the current terrain and state by choosing optimal parameters, instead of directly outputting motor position commands.

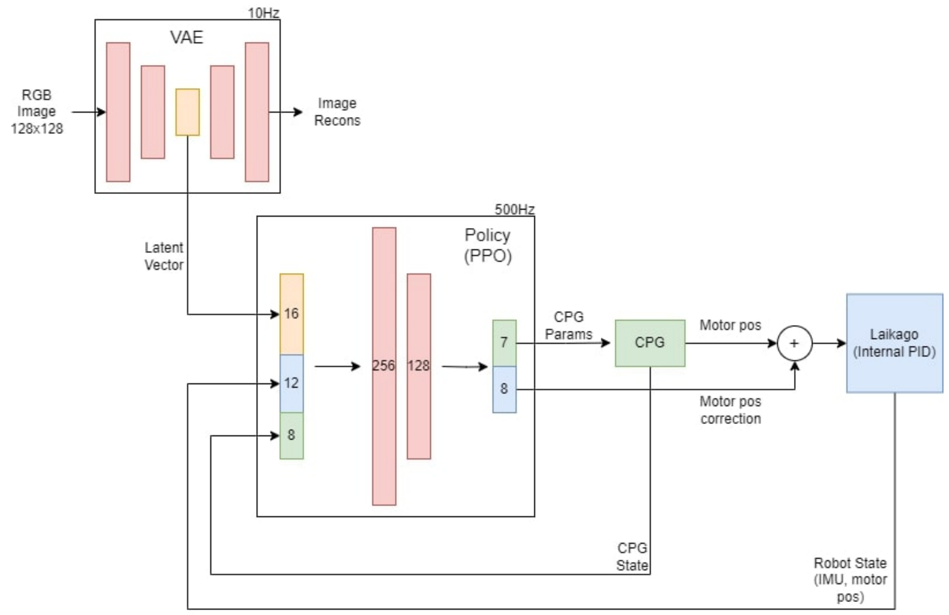


Figure 3.2: Gait control pipeline overview. Architecture consist of three modules: VAE for RGB image feature extraction, Fully Connected policy for determining optimal parameters and correction terms, and CPG to generate cyclical motor positions of the gait

In the following subsections we will go into further detail on the implementation of each module, their inputs and outputs, as well as their general role in controlling the robots gait.

3.2.1. Variational Autoencoder

The use of high-dimensional inputs, such as images, can lead to decreased sample efficiency in Reinforcement Learning scenarios. To address this challenge, it is useful to employ a model capable of encoding environmental features into a low-dimensional representation. A Variational Autoencoder comprises two connected networks, the encoder and decoder. The encoder learns to map the input into probability distribution parameters (μ, σ^2) used to sample a latent vector, a low dimensionality representation of the observed environment. The decoder then reconstructs the input image from the latent vector. Successful reconstruction indicates that the latent vector encapsulates all pertinent information from the input. For a more thorough description of VAEs refer to section 2.6 in the theoretical framework.

The simulated model of Laikago lacks inherent vision capabilities. However, we can address this limitation by employing PyBullet's getCameraImage function. This method allows for the capture of RGB images inside the simulation by specifying size, view, and projection matrices. Since we have access to the robot's true position and orientation at each timestep, we can leverage this information to calculate the matrices and acquire 128x128 RGB images capturing the viewpoint of the robot. The VAE takes these images as inputs and generates latent vectors, which are then fed into the next module. In our simulation running at 500Hz, a new latent vector is computed every 50 timesteps. This is primarily due to two reasons. Firstly, for computational efficiency, as the process of calculating matrices, obtaining the image, and passing it through the network can consume a considerable amount of time, potentially slowing down both the simulation and training. Secondly, the purpose of the VAE is to act as a gait planner in a hierarchical structure. Its objective is to interpret the environment and convey this information to the low-level control policy. It acts as a form of abstract command, providing guidance for the controller to interpret as the desired gait type for navigating the current observed environment. We opt not to select a new gait at every timestep, since we want them to be extended actions, and because the scenery does not change rapidly enough to have the need to. The abstract commands we strive to learn include instructions such as 'turn left', 'ascend the slope', and 'continue straight'. These commands encapsulate the higher-level commands we aim for the controller to interpret and execute in navigating the environment.

Network Architecture	**Layer Type**	**Output size**
Encoder	Input Layer	3x128x128
	Conv2D 1	32x64x64
	Conv2D 2	64x32x32
	Conv2D 3	128x16x16
	Conv2D 4	256x8x8
	Conv2D 5	512x4x4
	Dense μ	16
	Dense $\log(\sigma^2)$	16
Decoder	Dense	8192
	ConvTranspode2D 1	256x8x8
	ConvTranspode2D 2	128x16x16
	ConvTranspode2D 3	64x32x32
	ConvTranspode2D 4	32x64x64
	ConvTranspode2D 5	3x128x128

Table 3.1: Layers of the implemented variational autoencoder. Batch normalization and activation functions are used after every convolutional layer.

The architecture of the implemented VAE is detailed in Table 3.1. The encoder component consists of a sequence of five convolutional layers, designed to progressively increase the number of channels while simultaneously reducing the feature sizes. Following the fifth convolutional layer, the output is flattened and processed through two parallel dense layers, producing the μ and $\log(\sigma^2)$ parameters defining the latent distribution. We have chosen a latent vector size of 16, striking a balance between achieving commendable reconstructions and avoiding unnecessary dimensionality. The latent vector is sampled using the reparameterization trick. The decoder initiates by feeding the latent vector through a dense layer with an output size of 8192, subsequently reshaping it to a tensor of size 512x4x4. This reshaped tensor is then processed through a sequence of five transposed convolutional layers. These layers expand the size of features while simultaneously reducing the number of channels, contributing to the generation of the final output. After the last transposed convolutional layer, the input's shape is fully recovered. At each convolutional layer, whether normal or transposed, batch normalization and leaky ReLU activation layers are incorporated, except for the last layer where an hyperbolic tangent activation is used instead.

In our architecture image processing and motor control networks are trained separately, as they are part of different families of machine learning methods: unsupervised learning and reinforcement learning, respectively. While some studies have demonstrated the concurrent training of different levels in hierarchical structures [22], we opt to pre-train our VAE with a dataset of images taken directly from our custom environments. The utilization of VAEs for pre-training is motivated by their effectiveness in learning feature representations from images. VAEs have been shown to produce disentangled factors [35], indicating that each component of the latent vector corresponds to a specific feature.

We train using a dataset of 512 RGB images taken from the simulated environment. Out of these, 448 images are allocated for training, while the remaining 64 are set aside for validation. The training process spans 300 epochs, employing the hyperparameters detailed in Table 3.2. When loading data some transforms were applied to augment the dataset, adding more variation to the collected data by randomly flipping the images. As discussed in section 2.6, the objective function is a combination of reconstruction error and regularization term. A weight is added to the regularization to adjust its contribution to the overall loss. Since we are working with images we use Mean Squared Error between input x and output $\hat{x}$.

$$\text{Loss} = \text{MSE}(x, \hat{x}) + \text{KLD}_{weight} \cdot \frac{1}{2} \sum_{j=1}^{J} (1 + \log\left(\sigma_j^2\right) - \mu_j^2 - \sigma_j^2) \tag{3.1}$$

learning rate	kld weight	batch size	output activation	lowest loss
0.005	0.00025	32	Tanh	4135.13
0.005	0.001	16	Tanh	2188.98
0.005	0.00025	16	Tanh	2169.08
0.005	0.00025	16	Sigm	2123.24
0.0005	0.00025	16	Sigm	1956.96
0.0005	0.00025	16	Tanh	1945.38
0.005	0.00025	16	Tanh	1684.79

Table 3.2: VAE experiments hyperparameters and their lowest loss.

The determination of the lowest loss is based on evaluation images to mitigate the risk of overfitting to the training data. Results from our experiments indicate that the best reconstructions are obtained when employing a learning rate of 0.005 and a smaller batch size. While larger batch sizes were explored in our experimental setups, the low amount of available images led to significantly inferior results in those cases. It is noteworthy to mention that the lowest loss can vary considerably depending on the images on the validation set, therefore it is left as future work to acquire a larger dataset and retrain the model. Figure 3.3 showcases examples of reconstructions generated by our most proficient model. While the outputs successfully replicate the overall structure of the input, there are instances where the reconstructions may exhibit noise or generalize more unconventional scenes into more common ones.

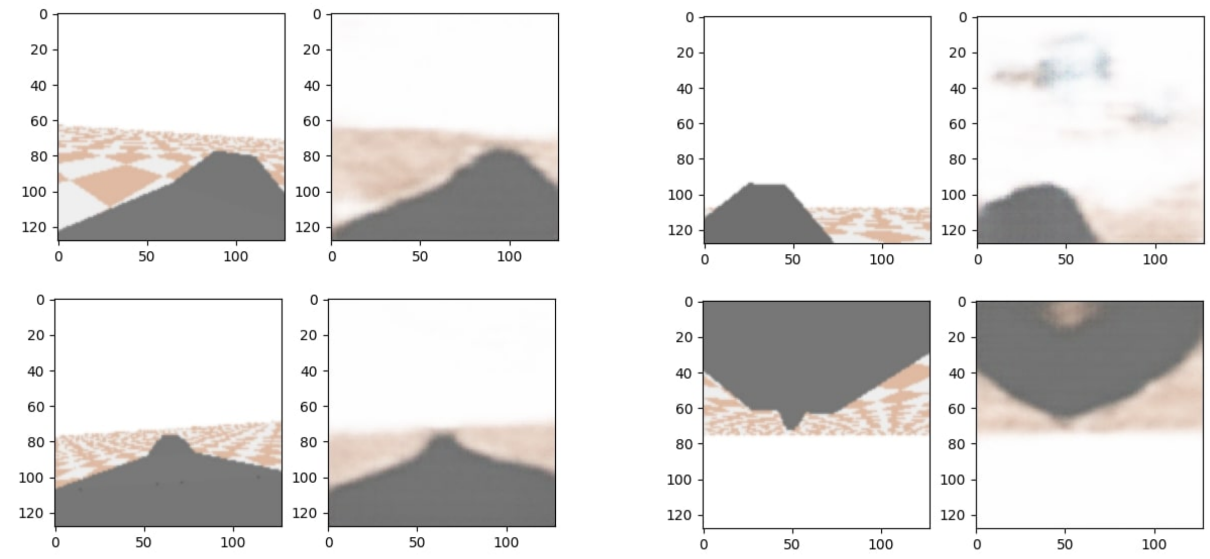


Figure 3.3: Examples of inputs (left) and their reconstructions (right)

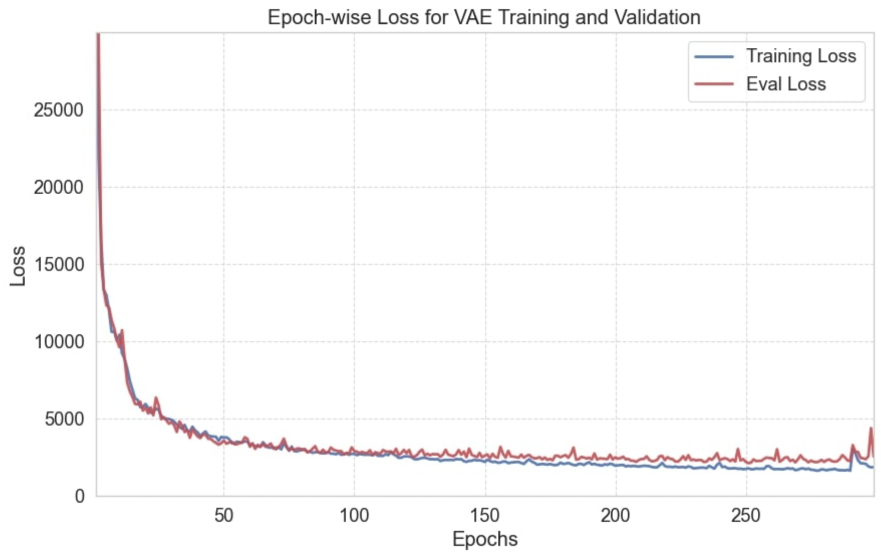


Figure 3.4: Training and validation loss over epochs for best VAE model

3.2.2. Policy Network

Our policy is implemented as a two-layered fully connected Multi-Layer Perceptron (MLP) network, trained using the PPO algorithm provided by Stable Baselines3. The input, recognized in RL as the observations of the current timestep, is a concatenation of several components: the latent vector obtained from the VAE, the Central Pattern Generators parameters and phase, and the robot's state. This input is then fed into the MLP network, which produces a set of seven parameters along with a correction term for each motor in use. While the Laikago robot has 12 onboard motors, our policy generates only 8 correction terms, specifically targeting the hip and knee motors (Figure 3.5). Notably, the coronal plane motor is intentionally excluded from the control signals. This deliberate restriction stems from the utilized CPG formulation, which models leg trajectories in a 2D plane. Consequently, the CPG does not account for movements perpendicular to this plane, resulting in the fixed positioning of the third motor.

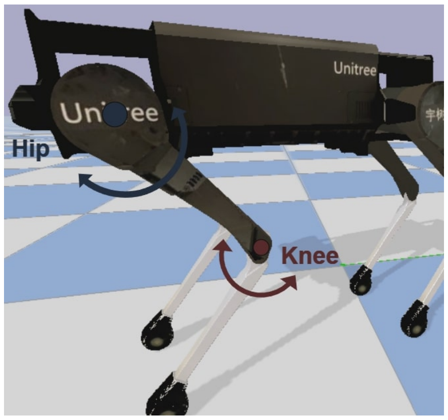


Figure 3.5: Degrees of freedom utilized by the controller. Blue is for hip motors, red for knee motors. Coronal plane motors have a fixed position

PPO follows the actor-critic methodology, involving the training of two networks: the policy and the value function. In our architecture, the presented MLP network serves as the actor, responsible for decision-making. The critic network, which estimates the value function, is not explicitly depicted, as its only relevant during the training phase. During training, the critic network is used for computing the advantage, guiding policy updates by assessing the quality of chosen actions.

We approach the task of gait parameter selection by formulating it as a Partially Observable Markov Decision Process (POMDP), where our robot possesses estimations of certain states. The segment of the input corresponding to the robot state contains information regarding orientation, angular velocities, and motor angles. For simplicity, we do not consider any measurement noise in these readings. Nevertheless, incorporating such noise could prove advantageous for enhancing the robustness of learned policies, particularly in scenarios where training is done in real hardware. The remaining segment of the input comprises the CPG's parameters that defined the legs trajectories during the last timestep, along with the current phase of the CPG. This information not only captures details about the form of the gait but also informs us about its position within the trajectory.

	Input	Description
Latent Command	Latent Vector (x16):	Features extracted from RGB camera image by Variational Autoencoder.
Robot State	IMU Readings (x4):	Roll, pitch and angular velocity in X, Y axes of the robot.
	Motor Positions (x8):	Current position of hip and knee motors.
CPG State	CPG Parameters (x7):	Central Pattern Generator parameters that define the trajectories to be followed by the legs: $f, A_h, A_{st}, A_{sw}, d, o_h, o_k$.
	CPG Phase (x1):	Phase of the oscillator.

Table 3.3: Description of the policy inputs

While Table 3.3 displays 36 observations, it is important to note that a diverse set of states and parameter combinations were tested. The configuration presented represents the outcome of a series of experiments aimed at identifying the optimal input. This considered both the relevance of individual observations and the overall size of the input. A larger input can result in less efficient sampling and heightened problem complexity. Therefore, our objective is to maintain a reduced input dimension. Following this principle, we actively avoid incorporating redundant information. For instance, we exclude yaw measurements, relying instead on camera image features to derive such information.

The output of the policy, conventionally regarded as the action, is not directly applied to the robot. The initial portion of the output comprises the new parameters for the CPG. To prevent abrupt changes in the trajectories generated by the CPG which could destabilize the agent, we implement a straightforward proportional controller to modify the parameter values. For instance, if we had frequency f_{old} last timestep and our output provided f_{new}, then we would update as follows:

$$f = f_{old} + \gamma(f_{new} - f_{old})\Delta t \tag{3.2}$$

We use a proportional constant γ to regulate the rate of change, and Δt is the length of a timestep. This process is done for each of the seven parameters. The policy runs at 500Hz, meaning if an instantaneous change to the values were desired, $\gamma = 500$ should be employed. Lastly, the correction terms outputed by the policy aim to adjust the motor positions generated by the CPG, adding residuals when needed. This is useful as that the CPG gives the same trajectory to all legs, but small differences are essential to allow behaviors such as steering and climbing. This approach of modulating trajectory generators has demonstrated success in learning intricate control behaviors, as evidenced by its creators in [36].

The ranges of values for both the observation and action spaces were deliberately selected to afford the policy the flexibility to explore diverse strategies, while simultaneously imposing constraints, ensuring that the policy operates within values that are sensible for the given problem. For instance, it is impractical to permit the hip motors to reach positions where the legs point upwards, or to have frequencies so high that no robot platform could feasibly follow. In defining these value ranges, we leverage our intuition about the problem and align them with practical considerations. While this strategy improves the efficiency of samples by excluding meaningless regions within the action space, caution must be taken to avoid overconstraining the action space. Overconstraint could introduce excessive bias, affecting the agent's ability to learn complex behaviors. The employed values for the observation and action spaces are shown in Tables 3.4 and 3.5, respectively.

	Observation	**Value Range**
Robot State	Roll, Pitch	$[-\pi, \pi]$
	ω_x, ω_y	$[-10.0, 10.0]$
	Hip motor (x4)	$[-1.0, 1.0]$
	Knee motor (x4)	$[-1.7, 0, 3]$
CPG State	Phase	$[0, 2\pi]$
	Frequency f	$[0.0, 10.0]$
	Amplitude A_h, A_{sw}	$[0.0, 1.0]$
	Amplitude A_{st}	$[0.0, 0.5]$
	Duty Factor d	$[0.5, 0.95]$
	Offset o_h	$[0.0, 0.4]$
	Offset o_k	$[0.3, 0.7]$
Latent Vector	Latent feature (x16)	$[-10.0, 10.0]$

Table 3.4: Observations value ranges

	Action	**Value Range**
CPG Parameter	Frequency f	$[1.5, 4.0]$
	Amplitude A_h, A_{sw}	$[0.0, 1.0]$
	Amplitude A_{st}	$[0.0, 0.5]$
	Duty Factor d	$[0.5, 0.95]$
	Offset o_h	$[0.0, 0.4]$
	Offset o_k	$[0.3, 0.7]$
Correction Term	Residual (x8)	$[-0.1, 0.1]$

Table 3.5: Actions value ranges

To assess the effectiveness of the actions undertaken by our agent and derive a gradient for optimizing policy weights, a well designed reward function is required. A good reward function encapsulates the core objectives of the task, guiding the agent by reinforcing actions that contribute to its success, and penalizing undesired behaviors. The reward function must strike a balance between shaping the agent's behaviors to align with task objectives and allowing room for exploration and discovery. We determine rewards by using the following piecewise equation:

$$r(t) = \begin{cases} 10[x(t) - x(t - \Delta t)](1 - \text{falling}) - |\omega_x| - |\omega_y| & \text{if not done} \\ -1000 + 1000t/t_{max} & \text{if done and } x(t) < \text{goal} \\ 0 & \text{else} \end{cases} \tag{3.3}$$

An episode is a sequence of interactions between the agent and the environment. It serves as a distinct trial during which our robot commences from an initial state and engages with the environment through a series of actions until a termination condition is met. Termination conditions could include reaching a predefined goal or encountering a state deemed as a failure. In response to the conclusion of an episode, the environment undergoes a reset, initiating a new episode and continuing the learning process.

The first case in Equation 3.3 applies when an episode is still ongoing. We give out a reward proportional to the forward displacement of the robot's base since the last timestep, also acting as a penalization when moving backwards. This term is nullified if the robot is deemed to be falling, determined by its orientation and angular velocities. This adjustment prevents converging to suboptimal policies, such as rapidly falling forward to receive an immediate large reward but ending the episode prematurely. Additionally, we introduce penalties for angular velocities ω_x and ω_y, as these can be interpreted as indicators of potential instabilities.

Each episode is limited to a maximum duration of t_{max} timesteps but can conclude prematurely if the agent achieves a predefined goal or is deemed to have failed. In our implementation, failure is triggered if the robot falls or moves back beyond a specified distance, determined by evaluating the robot's true orientation and position provided by the simulator. The second case of our reward function imposes a penalty when an episode ends before reaching the goal. To account for variations in performance among failed trials, we make this penalty inversely proportional to the duration of the episode. In other words, the longer it takes for the episode to fail, the milder the punishment.

Finally, if the goal is reached, the episode concludes, and a reward of zero is assigned. We refrain from providing a bonus for reaching the goal, as unsmooth reward functions have the potential to introduce instability in the learning process.

3.2.3. Parameterized CPG Gait

Exploring the complete action space of the 12 onboard motors of the Laikago can lead to a slow learning process, particularly since the legs start off with no sense of coordination. Introducing prior knowledge has the potential to expedite training by reducing the number of samples required to acquire essential skills. This synergy between prior knowledge and reinforcement learning is especially well-suited for robotic locomotion, given the abundance of model-based controllers traditionally employed in this domain. Notably, Central Pattern Generators are widely used for generating repetitive, cyclic movements.

In our approach, we constrain the movements of the Laikago's hip and knee motors to follow periodic trajectories defined by Equations 3.4 through 3.10. These equations, derived from [37], parameterize a gait specific to quadruped locomotion, providing a structured foundation for learning locomotion skills in a more efficient manner.

Hip motor equations:

$$\Theta_i = (2\pi f \Delta t + \frac{\pi}{2} i) \bmod 2\pi, \quad i \in [0, 1, 2, 3] \tag{3.4}$$

$$\phi_i^h = \begin{cases} \frac{\phi_i}{2d} & \text{if } \Theta_i^h < 2\pi d \\ \frac{\phi_i + 2\pi(1-2d)}{2(1-d)} & \text{else} \end{cases} \tag{3.5}$$

$$\theta_i^h = A_h \cos \phi_i^h + o_h \tag{3.6}$$

Knee motor equations:

$$A_{k,i} = \begin{cases} A_k^{st} & \text{if } \phi_i^k < \pi \\ A_k^{sw} & \text{else} \end{cases} \tag{3.7}$$

$$\phi_i' = 2(\frac{\phi_i^k}{2\pi} \bmod 0.5) \tag{3.8}$$

$$\Gamma_i = \begin{cases} -16\phi_i'^3 + 12\phi_i'^2 & \text{if } \phi_i' < \frac{1}{2} \\ 16(\phi_i' - \frac{1}{2})^3 - 12(\phi_i' - \frac{1}{2})^2 + 1 & \text{else} \end{cases} \tag{3.9}$$

$$\theta_i^k = A_{k,i}\Gamma_i + o_k \tag{3.10}$$

Four coupled oscillators ϕ_i with frequency f govern each one of the robot's legs, as shown in Figure 3.6. There is a fixed sequential 90° phase shift between oscillators going in order: front left (FL), back right (BR), front right (FR), back left (BL). The gait is split into two stages: stance and swing. The ratio of the duration of each stage to the cycle is determined by the duty factor d, where a value close to 1 gives a longer stance and shorter swing, and vice versa for value close to 0. Equation 3.5 transforms the phase in order to extend/shorten the stages duration in accordance with the duty factor. The target motor angle of the hip θ_i^h is then calculated as per Equation 3.6, where A_h is the hip oscillator amplitude, o_h is an offset which acts as the continuous component or center of oscillation and ϕ_i^h is the transformed phase of the hip.

The radius of the knee oscillators is determined by the current stage of the gait. In general, the stance has small movements as the leg remains extended, while the swing contracts the leg with a broad movement to lift it from the ground. We include this prior knowledge by having a reduced value range for the stance amplitude, but ultimately leave it to be discovered by the agent during training. The knee phase aligns with the hip phase, although an optional shift $\phi_k = \phi_h + \psi_{hk}$ can be used. The piecewise cubic profile described by Equation 3.9, where ϕ_i' denotes a transformed version of the knee phase resized to the interval $[0, 1]$, serves as a tool for defining a smooth trajectory for the knee joint profile.

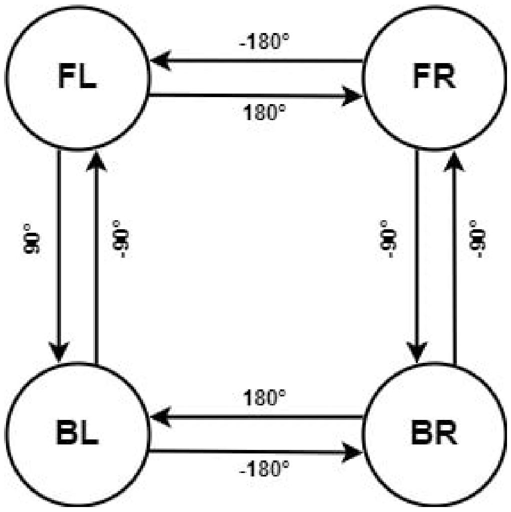
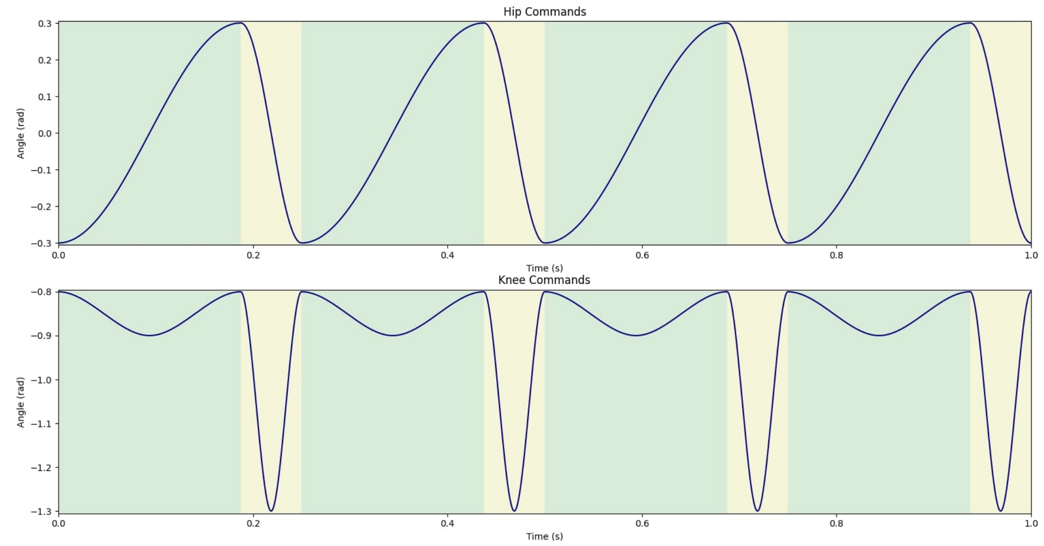


Figure 3.6: Phase shift between coupled oscillators

The cubic profiles produce a reciprocal motion in the knees, oscillating between swing and stance phases for each stage's duration. The calculated motor positions, denoted as θ_i^k, are derived using Equation 3.10, where the profile is scaled by the amplitude of the current gait stage and an offset o_k is added. Utilizing these equations with a predefined set of parameters results in the generation of periodic leg movements, exemplified in Figure 3.7. By smoothly varying parameters, the gait's form can be dynamically altered to allow the robot to traverse a changing environment.



.

Figure 3.7: Output of motor commands given by the CPG. Green and yellow areas indicate stance and swing respectively. $f = 4$, $d = 0.75$, $A_h = 0.3$, $A_k^{st} = 0.1$, $A_k^{sw} = 0.5$

3.3. Step-by-Step Process

To develop and train our control system, we established a systematic series of steps. The pipeline is outlined in Figure 3.8, acknowledging that certain steps may necessitate revisitation or further tuning.

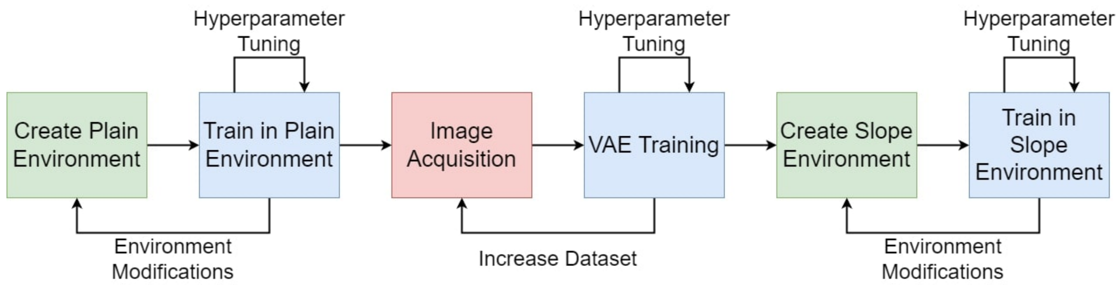


Figure 3.8: Step-by-Step procedure followed in this work

3.3.1. Plain Environment

Our primary goal is to train a quadruped robot to learn walking behaviors, enabling it to navigate a narrow path of varying slopes. However, before trying our approach on challenging environments, we deem it crucial to validate its efficacy in the simplest scenario first. If the robot struggles to traverse a flat surface successfully, most certainly it won't be able to learn to adapt to slopes. We begin by setting up a plain environment for our agent to learn a a straightforward walking-forward behavior. In this context, we find it unnecessary to employ the VAE module of our architecture. The environment remains constant, and any orientation information that could be derived from the camera is already contained in the input as part of the robot's state.

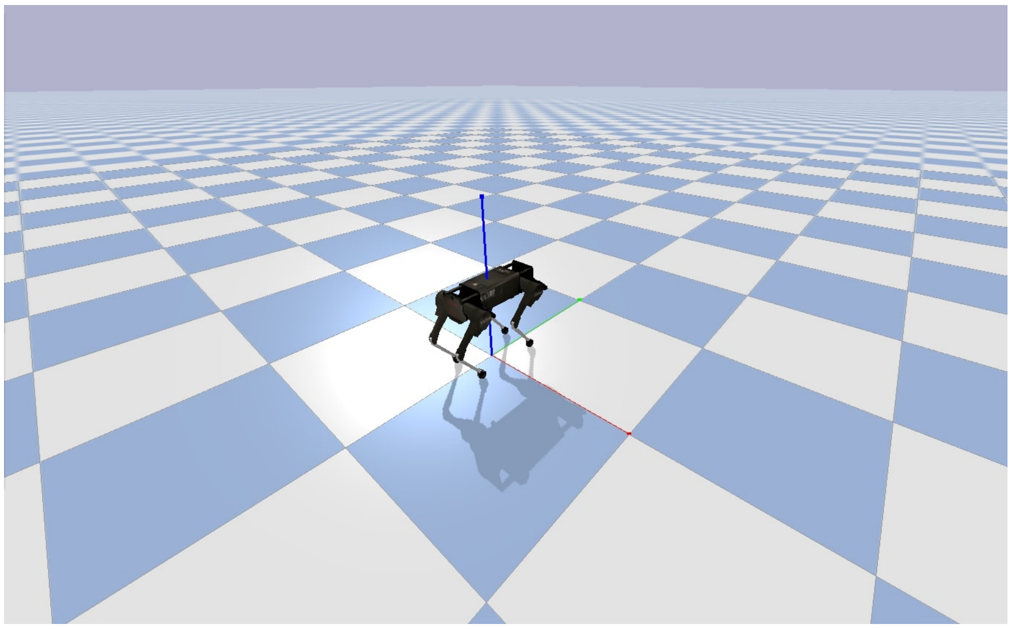


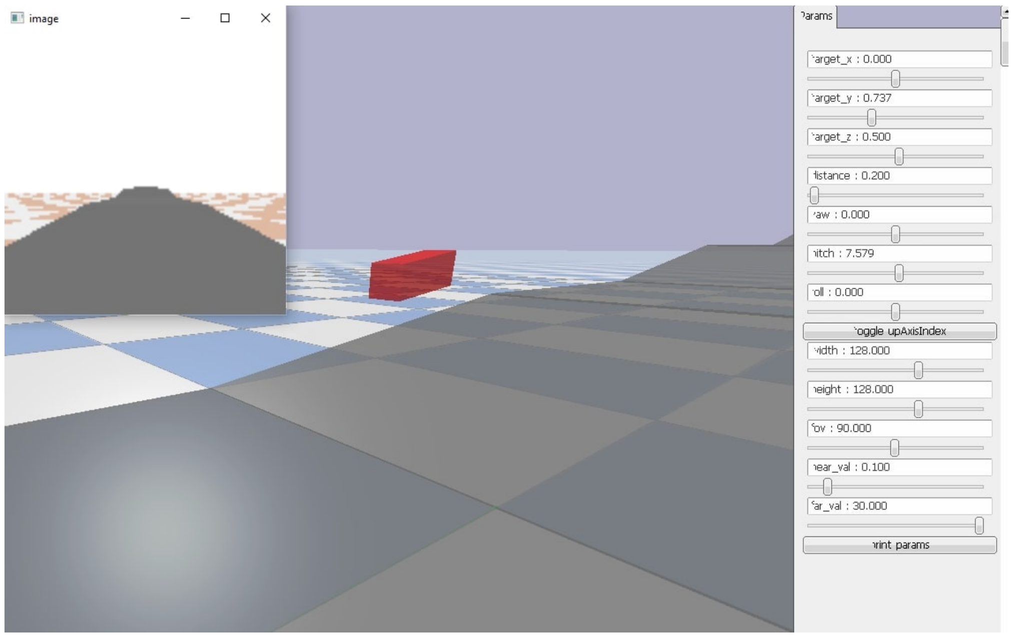
Figure 3.9: Plain Environment

The observation and action spaces configurations were determined through experimentation involving various versions of the environments. This included adding or removing observations, adjusting value ranges, and modifying the CPG formulation. Throughout this process, different sets of hyperparameters were employed during training to identify configurations that produced optimal results. These hyperparameters included the number of steps between policy updates, minibatch size, learning rate, total number of timesteps, activation function, and the number of neurons per hidden layer. After reaching the final configuration of the plain environment, further hyperparameter tuning was conducted.

While noise measurements were not considered in this work, we introduced random CPG parameter values during the initialization of episodes. This was implemented to train more robust policies capable of handling a diverse range of starting conditions. This randomization aspect was applied not only in the plain environment but also in the slope environment, which will be discussed next.

3.3.2. Image Acquisition

For experiments conducted in a more complex environment, the vision module becomes essential for extracting information about what the robot perceives. VAEs, being an unsupervised learning method, require a substantial dataset of unlabeled data to be effectively trained. However, given that our robot lacks the capability to autonomously navigate through the more complex environment at this point, relying on it for data collection is not feasible. To address this limitation, we developed a program that enables manual repositioning of a camera within the simulation. This allows us to choose the camera's position and orientation for data collection purposes.



arget_x : 0.000
arget_y : 0.737
arget_z : 0.500
istance : 0.200
aw : 0.000
itch : 7.579
oll : 0.000
idth : 128.000
eight : 128.000
ov : 90.000
ear_val : 0.100
ar_val : 30.000

Figure 3.10: Image acquisition program. The red box represents the position of the camera in the simulation, its point of view is seen in the pop-up window. On the right, camera position and orientation can be controlled

At the program's initiation, a path of randomly generated slopes is created, in the same process later to be used in the slope environment. This ensures the captured environment will not be the same for all captured images. To assemble a varied dataset, the camera was moved to different positions, capturing a wide range of perspectives. During intervals in the data acquisition process, VAE models were trained to assess performance with the current image count. This approach not only helped determine if the dataset was sufficiently large but also aided in identifying overfitting or potential lack of representation for certain image types. Upon collecting a total of 512 RGB images, the final VAE model was attained through training and hyperparameter tuning, as illustrated in Table 3.2.

3.3.3. Slope Environment

Having trained a VAE model that returns acceptable reconstructions of input images, we proceed to learn a policy with our complete architecture on a more complex environment. At the start of each episode, we generate a path for the robot to traverse. This path consists of a sequence of alternating ramps and platforms. The slope of each ramp is randomly sampled from a range of 5 to 10 degrees, and the length of each segment can vary, spanning 2, 3, or 4 meters.

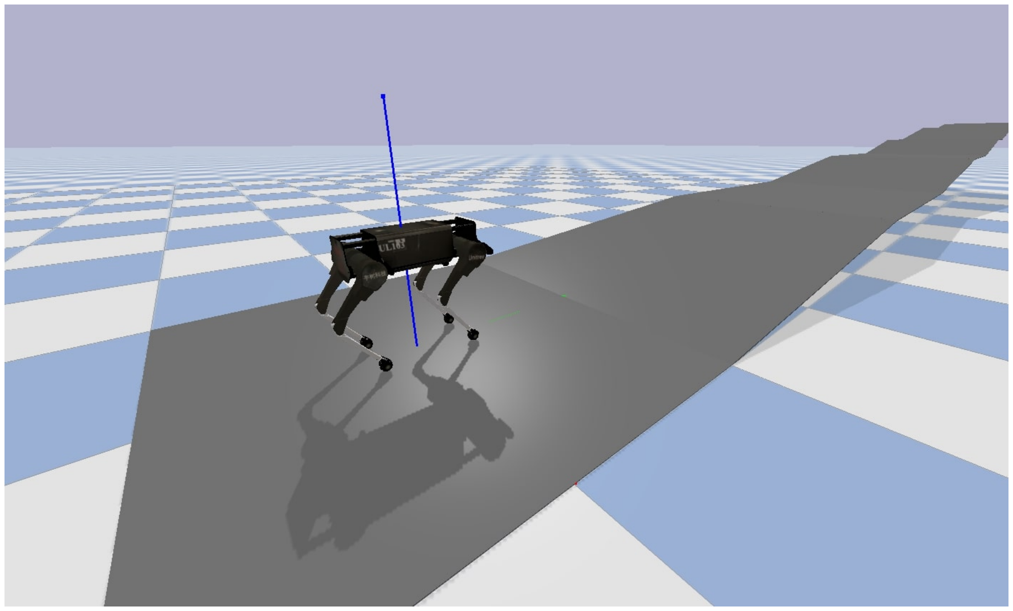


Figure 3.11: Slope Environment

Several iterations of the environment underwent testing, requiring consideration of previously absent factors. One crucial aspect involved selecting an appropriate inclination range for ramp segments. Excessively steep slopes posed the risk of the agent slipping, hindering forward progress. Conversely, reducing the slope inclinations too much could oversimplify the problem, potentially leading the agent to learn a single gait suitable for traversing all segments, rather than dynamically adapting its walking behavior. Similarly, different friction coefficients were tested. Although opting for a high friction coefficient might seem advantageous to prevent slipping, we chose a more realistic value of 0.5, akin to surfaces found in roads, soil, and wood. Lastly, recognizing that a more complex environment may demand a broader range of movements, we fine-tuned the action and observation spaces to provide the agent with more flexibility. We then returned to our plain environment to train with the finalized configurations, as presented in Tables 3.4 and 3.5.

Chapter 4

Experiments and Results

In our experimental setups, our primary objective is to validate the efficacy of our approach by comparing it to the performance of a state-of-the-art algorithm. To accomplish this, we have opted to employ a policy trained with PPO. This policy is designed to directly select motor commands without relying on any prior knowledge from a CPG. By comparing both methods, our objective is to showcase the benefits of incorporating prior knowledge in the learning of quadruped robot walking behaviors. Furthermore, we assess the performance of a modified version of our architecture in which we have eliminated the modulation of the CPG's outputs. This is done to analyse the modulation's impact on the generated gaits and the overall success of the task. For each environment (plain and slope) we present three experimental setups:

Method	**Description**	**Obs dim**	**Act dim**
PPO	Policy that directly chooses motor positions, trained using the Proximal Policy Optimization algorithm	12 + latent dim	8
PPO+CPG	Policy that chooses parameters of a Central Pattern Generator, which in turn generated periodical motor positions	20 + latent dim	7
PPO+CPG+mod	Incorporates modulation of the CPG motor position output by addition of correction terms chosen by the policy	20 + latent dim	15

Table 4.1: Experiment methods description

Note that "latent dim" refers to the observations given by the VAE, and only applies to the experiments done in the slope environment. While training the three methods in the same environment, we maintain consistent hyperparameters. However, these hyperparameters may not be identical between the plain and slope configurations. Each set of values is fine-tuned to optimize the overall reward while training with the complete architecture.

4.1. Plain environment experiments

The selected hyperparameters are outlined in Table 4.2. The training process spans a total of 10 million simulation steps, with the simulation running at a frequency of 500Hz. This translates to approximately 5 and a half hours of simulation time. We sample trajectories for n_steps between policy updates and utilize batches of size batch_size to execute these updates. Our actor and critic networks, although they do not share weights, possess an equal number of hidden layers and neurons. The hyperparameter gamma refers to the proportional constant γ used to update the CPG's parameters values in Equation 3.2. Lastly, freq_range indicates the range of values the actor can choose from for the oscillators frequency. This is the only CPG parameter which value range we specify in the initialization of an experiment, as it showed the biggest impact on the strategies learned by the agent.

Hyperparameter	**Value**
n_steps	16384
batch_size	2048
lr	0.0003
tot_timesteps	10^7
activation_fn	Tanh
actor_arch	[200,200]
critic_arch	[200,200]
gamma	10.0
freq_range	[1.5,4.0]
max_episode_lenght	5000

Table 4.2: Hyperparameter values for plain environment experiments

Figure 4.1 shows a comparison between the reward curves obtained during the training of the three methods. Our approach, both with and without modulation, vastly outperforms the policy obtained by using only the PPO algorithm in terms of the reward obtained by the agent. The non-modulated method's reward rises the fastest at the start of training, and reaches the overall highest reward. And while the modulated method falls behind, it isn't by a large margin and achieves a similar performance. The difference in sample efficiency and overall reward can be attributed to the introduction of correction terms, which increases the dimension of the action space that the agent explores. It could also be because of the simplicity of the problem. Walking through a flat surface doesn't require adapting the gait to obstacles or changes in the terrain, so the modulation of the CPG's outputs might not be necessary.

To enhance our understanding of the performance of each method, we conducted 50 episodes with the trained models. We collected data on various metrics, including the distance covered by the agent before episode termination, the number of successful attempts at reaching the goal (defined as 10 meters from the starting position), and the corresponding durations for each successful attempt. Figure 4.2 presents a histogram depicting the final positions of the agent at the conclusion of each episode. This visualization provides insights into the distribution of final positions across multiple trials.

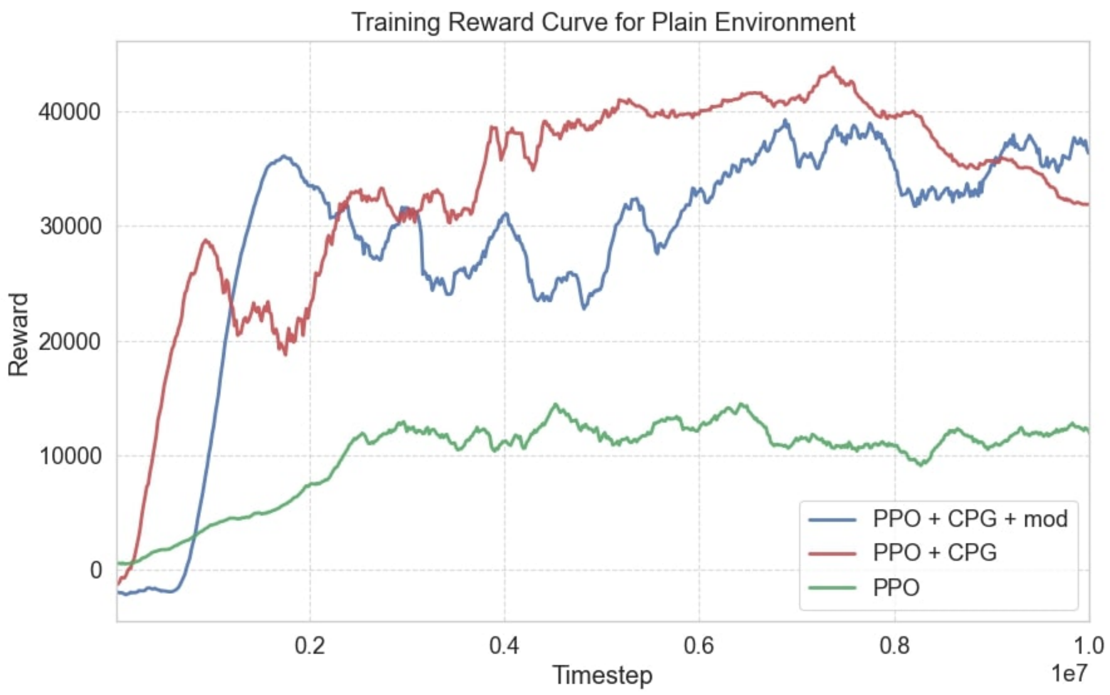


Figure 4.1: Comparison of rewards obtained during training in plain environment for three methods

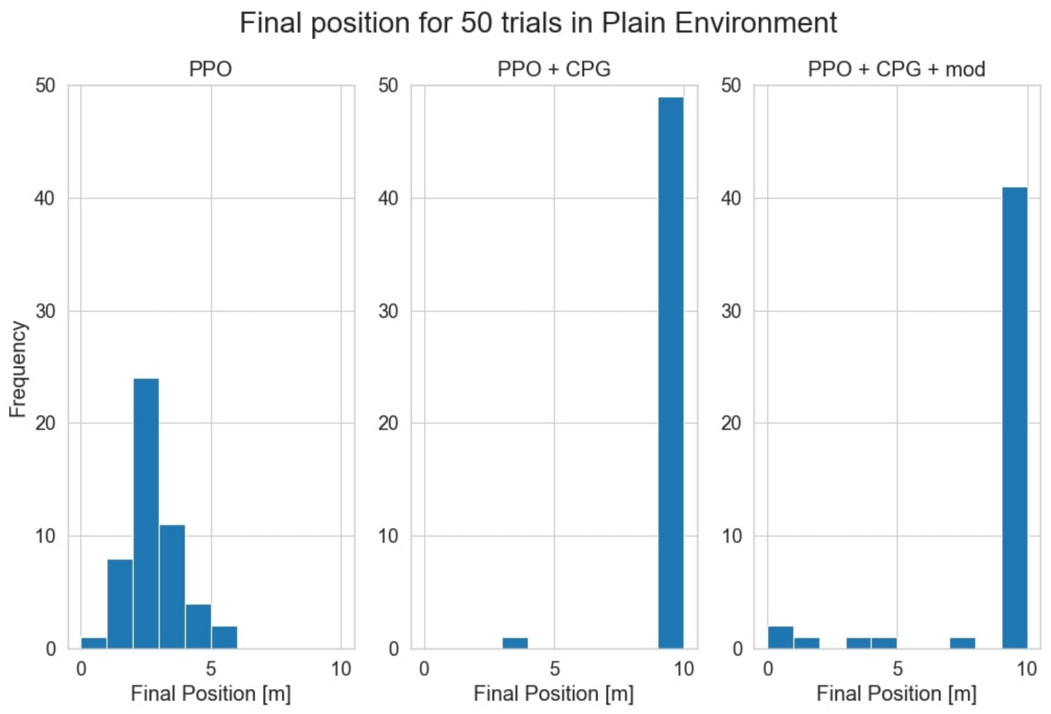


Figure 4.2: Histogram of final position reached by the agent in the plain environment for three methods.

Based on these results, it is evident that the agent struggles to make substantial progress when relying solely on the PPO algorithm for decision-making. Notably, it fails to surpass a distance of 6 meters in any of the conducted trials. In contrast, the non-modulating approach exhibits great consistency, surpassing the 9-meter mark in almost every trial. On the other hand, modulating the motor commands shows slightly less consistency, encountering failures at various points in the trajectory. Nevertheless, it still achieves success in the majority of trials.

These findings align with the reward curves obtained, suggesting that the second method demonstrates superior performance. The performance observed in both the distance covered and the reward curves indicates the effectiveness of the non-modulating approach in comparison to the modulating method in plain environments. However, upon reviewing the results presented in Table 4.3, an advantage of the latter becomes apparent. While it may reach the goal less frequently, the modulating method achieves this with higher speed, evidenced by a mean goal time lower than that of the non-modulating alternative. Given that we both reward the velocity of forward progress and penalize early termination, a trade-off between speed and stability emerges between the two methods.

Method	**Mean final pose [m]**	**Mean goal time [s]**	**Success rate**
PPO	2.85	N/A	0%
PPO+CPG	9.88	11.92	98%
PPO+CPG+mod	8.54	9.72	78%

Table 4.3: Performance metrics of 50 trials in plain environment

Utilizing our architecture, the agent demonstrates the ability to learn behaviors allowing for forward progress across flat terrain, successfully reaching a 10-meter goal within a reasonable timeframe. To gain insights into the learned behavior, we record the values of the CPG's parameters during a successful run, specifically employing the modulating method. To enhance clarity, we present the parameters in three distinct plots, as illustrated in Figure 4.3. Immediately it becomes evident that the policy doesn't merely learn an optimal value for each parameter and keep it fixed. Instead, it learns to output periodic signals for each parameter, continually varying them to achieve a walking behavior. The only exceptions are the amplitude of the knee motor during stance and the hip offset. At the start of the episode the values of each parameter are randomly chosen, but are rapidly pushed to the optimal value range by the policy. The signals remain relatively consistent throughout the entire episode, likely due to the unchanging nature of the environment the agent traverses.

Finally, we also recorded the angles of the 8 utilized motors during the successful run, as depicted in Figure 4.4, were we have separated between forward and back legs for our plots. These results reflect the walking gait defined by the CPG formulation. We can identify broader movements in the hip motors, as well as the two phases of stance and swing in the knee motors. While the trajectory of the motors may not perfectly resemble the one shown in Figure 3.7, this discrepancy is expected. The parameters are under constant change, and the actual position of the motors is also influenced by external forces applied to the joints. Phase shifts between legs are evidenced in the trajectories. Specifically, there are $\pm 180°$ shifts between left and right motors, and $\pm 90°$ shifts between front and back.

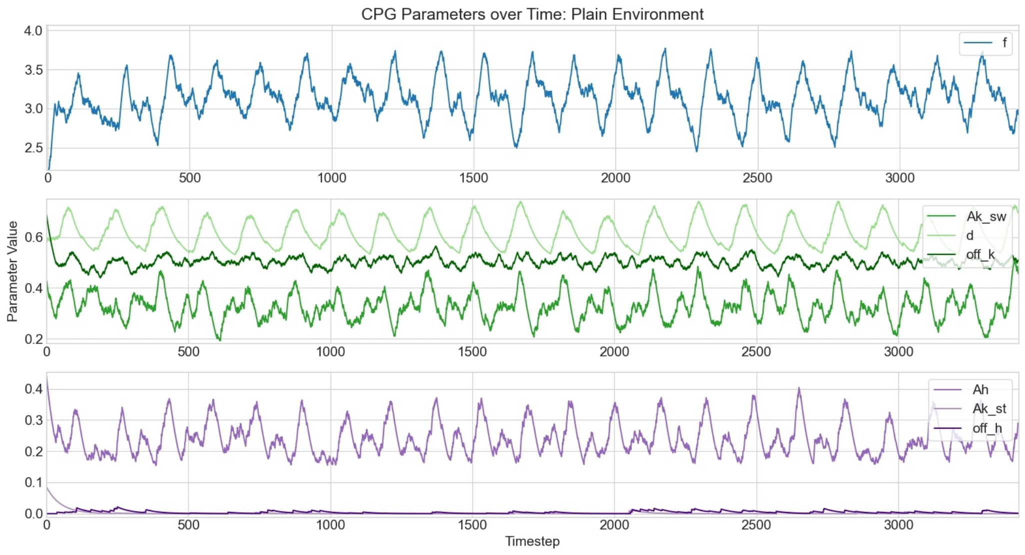


Figure 4.3: CPG parameters over time in a successful run in plain environment, using PPO + CPG + mod method

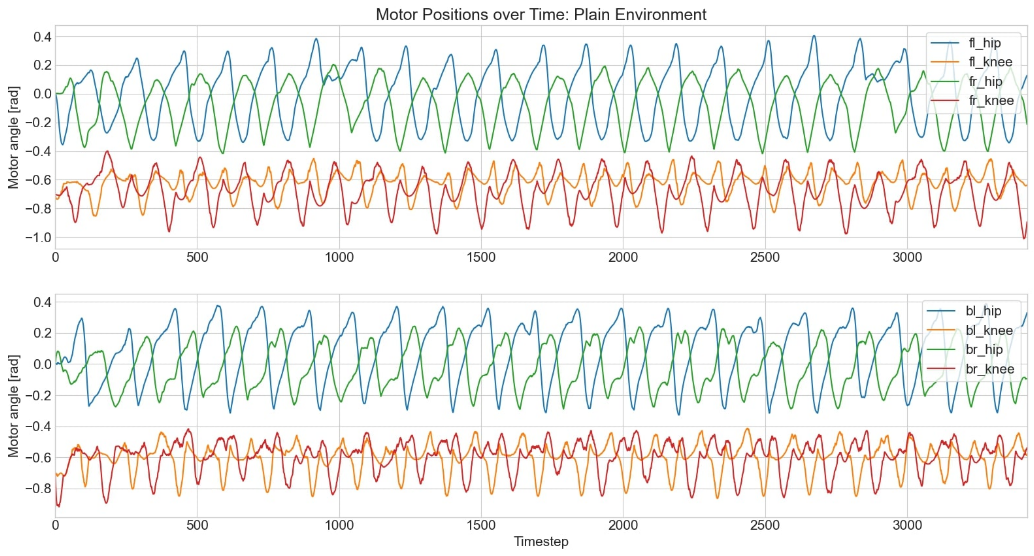


Figure 4.4: Motor positions over time in a successful run in plain environment, using PPO + CPG + mod method

4.2. Slope environment experiments

For this more complex task, we modify the hyperparameters to increase the amount of trajectories sampled between policy updates, and incriminating the batch size. This should give us a better estimate of the gradient, but to avoid divergences during training, we also decrease the learning rate. Additionally, we make alterations to the network architecture by expanding the number of neurons in the first hidden layer and decreasing them in the second layer. This type of configuration, known as “funneling”, is often used with the purpose of extracting features in a hierarchical manner, which could prove useful when dealing with a higher dimensionality input.

Hyperparameter	**Value**
n_steps	65536
batch_size	8192
lr	0.0002
tot_timesteps	10^7
activation_fn	Tanh
actor_arch	[256,128]
critic_arch	[256,128]
gamma	10.0
freq_range	[1.5,4.0]
latent_dim	16

Table 4.4: Hyperparameter values for slope environment experiments

Once again, we evaluate the rewards achieved by the three methods throughout a training period of 10 million timesteps, as illustrated in Figure 4.5. It is evident that Vanilla PPO struggles to learn high-return behaviors within the designated timeframe. Upon simulating the trained policy, it becomes apparent that the algorithm converges towards a suboptimal local maximum. The learned strategy involves a rapid forward fall, resulting in a substantial reward during the initial timesteps of an episode, but leads to an early termination right after. Examining the first histogram in Figure 4.6, we can see that the distribution of final positions for the robot is centered around 2 to 3 meters, rarely finishing outside this range. From the smooth increment seen in the reward curve we can assume it learned and refined this strategy over time.

The non-modulated CPG method exhibits a swift rise in rewards initially but ultimately converges to a suboptimal policy around 30,000 timesteps. Despite this, it performs well in terms of forward displacement, with over half of the trials successfully reaching the goal and achieving an overall mean final pose of 8.97 meters (Table 4.5). The difference in rewards arises from the extended time required by the agent to reach the goal, lagging 7 seconds behind the modulated method. During trials with the trained policy, it is evident that the Laikago advances cautiously with tight movements, adopting a safer gait to avoid early falls. While this strategy is not inherently incorrect, it tends to be suboptimal and leads to convergence to lower rewards. The observed behavior can be attributed to a specific aspect of our reward function outlined in Equation 3.3. In cases of early termination, a penalty is imposed, but this penalty diminishes proportionally to the duration it took for the episode to conclude. Consequently, falling very early in an episode is considered more detrimental than falling later on. To address such scenarios, a modified version of the reward function could be proposed. Instead of diminishing the penalty based on the length of the episode, the modification would involve basing it on the proximity of the agent to the goal:

$$r(t) = \begin{cases} 10[x(t) - x(t - \Delta t)](1 - \text{falling}) - |\omega_x| - |\omega_y| & \text{if not done} \\ -1000 + 1000 \cdot \text{current\_pos/goal} & \text{if done and } x(t) < \text{goal} \\ 0 & \text{else} \end{cases} \tag{4.1}$$

Lastly, the CPG modulated method receives the highest reward, and its good performance is evidenced by the results of the trials conducted after training, as shown in Figure 4.6 and Table 4.5. The agent reached the goal 74% of the time, averaging a mean final pose of 9.07, surpassing the two previous methods. Notably, the controller exhibits robustness in adapting to randomized starting conditions, as the robot avoids any falls within the initial two meters of the trajectory. Although proficient walking behaviors were acquired, it is noteworthy that there is room for improvement in sample efficiency. The introduction of modulating factors for each motor affected the learning speed of the agent, as evidenced by the fact the optimal version of the model was achieved only after nearly 10 million simulation steps.

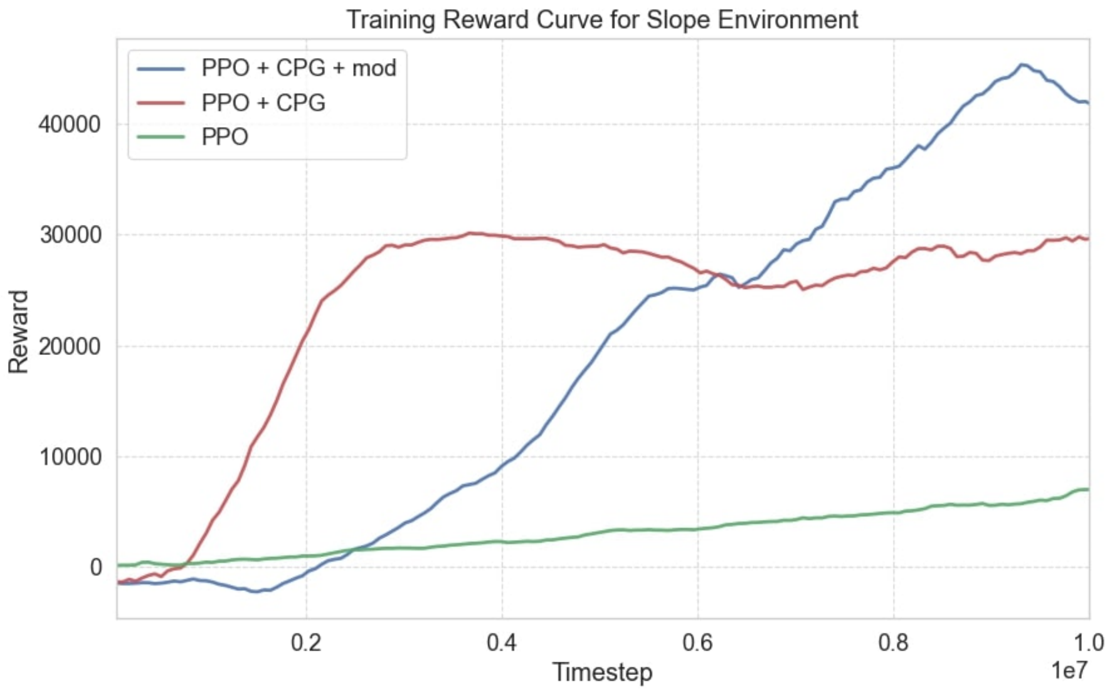


Figure 4.5: Comparison of rewards obtained during training in slope environment for three methods

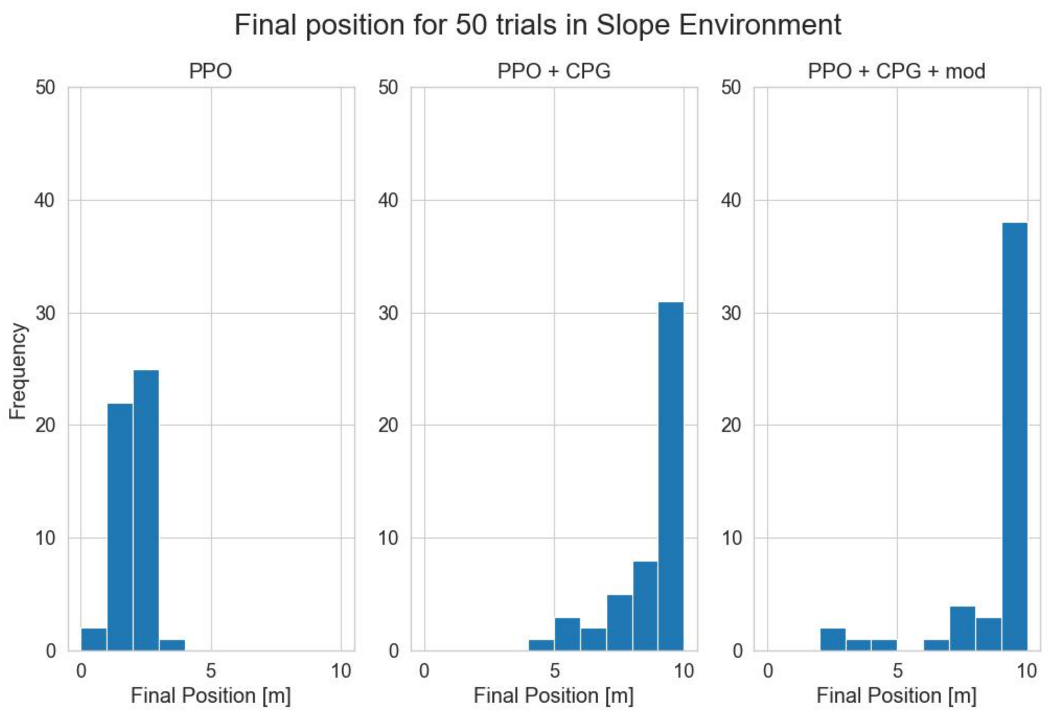


Figure 4.6: Histogram of final position reached by the agent in the slope environment for three methods.

Method	**Mean final pose [m]**	**Mean goal time [s]**	**Success rate**
PPO	2.02	N/A	0%
PPO+CPG	8.97	28.17	56%
PPO+CPG+mod	9.07	21.02	74%

Table 4.5: Performance metrics of 50 trials in slope environment

To investigate whether our agent adapts its gait in response to terrain observations, we visualize the CPG parameters and motor positions. A colored background is added to denote the type of surface the robot is traversing at each moment. White signifies a platform with no inclination, and gray a ramp with a slope of 5-10 degrees. In Figure 4.7, it is observed that the plain segments of the trajectory generally exhibit smaller frequencies and amplitudes in the oscillators, while the other parameters remain relatively constant across both scenarios. The most significant variations in parameter values occur during transitions between different terrain types. This observation aligns with the critical nature of these transition points, where the robot must adapt its footing to prevent potential falls. Once a transition is successfully navigated, the parameters tend to stabilize until the next one. This behavior is also reflected on the motor positions shown in Figure 4.8. The biggest changes in their movement pattern happen when a change of terrain is close to happening. Notably, there is noticeable variation in the positions of the front legs, which undergo more drastic changes. This divergence is attributed to their role as the initial contact points with the new terrain. The distinct movement patterns between front and back legs are made possible by the incorporation of modulating factors. Without these factors, all legs would follow the same periodic joint trajectory.

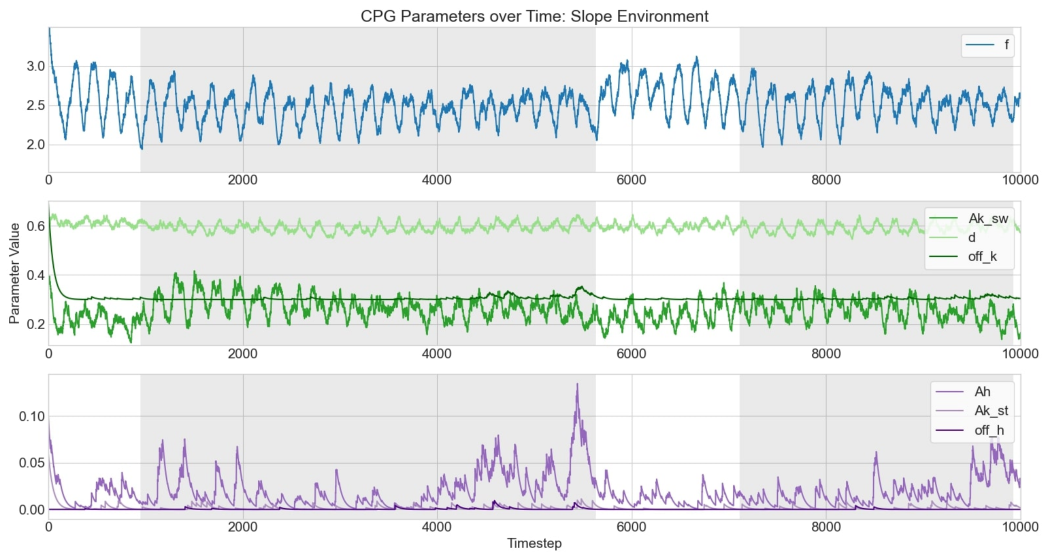


Figure 4.7: CPG parameters over time in a successful run in slope environment, using PPO + CPG + mod method. White area: Horizontal platform. Gray area: Ramp with slope

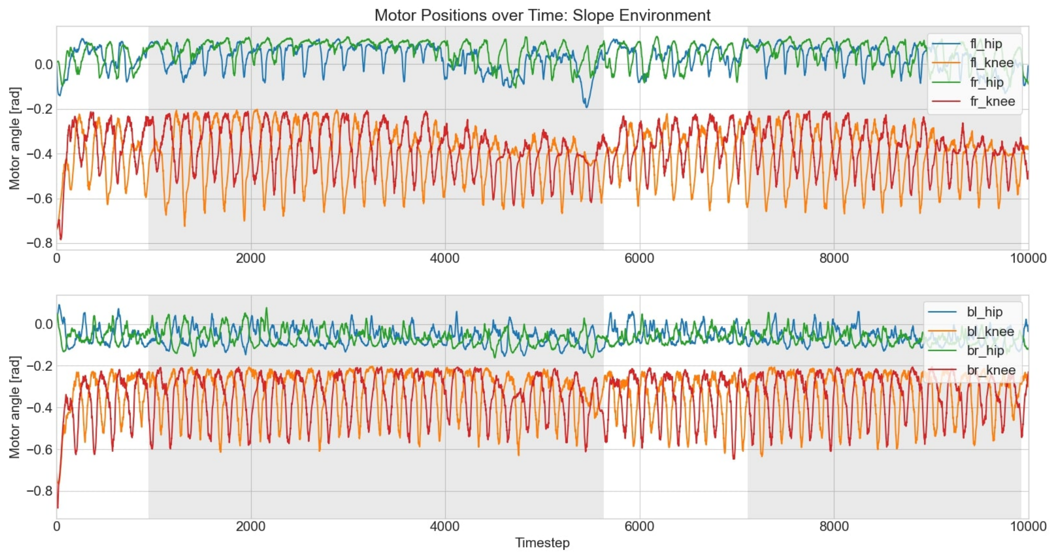


Figure 4.8: Motor positions over time in a successful run in plain environment, using PPO + CPG + mod method. White area: Horizontal platform. Gray area: Ramp with slope

In the previous slope environment experiments, we employed an extensive number of steps for each update in the training process. While this approach offers advantages, such as providing a more accurate gradient estimate and enhancing the model's generalization through exposure to a diverse set of samples, it comes with the trade-off of significant computational resource consumption and extended training times. Additionally, a fundamental challenge with the PPO algorithm is its strong dependence on hyperparameter choices during training, which can greatly affect performance. Considering these factors, we further tested our methods using a different set of hyperparameters, as listed in Table 4.6. We opted for smaller policy updates and batch sizes, as well as the network architectures used for the plain environment experiments

Hyperparameter	**Value**
n_steps	16384
batch_size	1024
lr	0.0003
tot_timesteps	10^7
activation_fn	Tanh
actor_arch	[200,200]
critic_arch	[200,200]
gamma	10.0
freq_range	[1.5,4.0]
latent_dim	16

Table 4.6: Hyperparameter values for slope environment experiments with smaller policy updates

The reward curves in Figure 4.9 show that the simple PPO approach still performs very poorly, unable to obtain any sort of meaningful reward. This is further evidenced by the post-training trials data, indicating that the agent never progresses beyond 1 meter, with a mean final pose of only 0.22 (Figure 4.10, Table 4.7). Moreover, in some episodes, it even falls backward, which explains why the histogram does not sum up to 50 trials, as negative final positions are not displayed. Across all experiments, it becomes evident that employing PPO to directly control motor positions is not a viable strategy. The combination of a large action space and low sample efficiency poses significant challenges for the agent to learn walking behaviors effectively. Although it might be conceivable to train these behaviors through parallel training of multiple quadrupeds, with a well-defined reward function and an extensive number of sampled trajectories, this approach fundamentally contradicts the objectives of this work. The primary aim is to train quadruped robots to walk efficiently, rather than relying on brute force methods.

The new hyperparameters enabled the non-modulated method to acquire walking behaviors that successfully accomplish the task, avoiding convergence to suboptimal solutions observed in previous experiments. Its highest rewards are on par with the modulated method, but requires more episodes to be reached. This observation suggests that the performance made possible by the advantages provided by the correction terms may be replicated by the simpler system with sufficient training and appropriate hyperparameter selection. Mean-while, the modulated method demonstrates greater efficiency in learning the adaptable gait compared to the previous experiment shown in Figure 4.5. The optimal versions of the model are achieved at approximately 5 million steps, equivalent to around 3 hours of simulated time. This suggest that the complexity of the problem does not require massive amount of samples per update, and good policies can be achieved with smaller, more frequent updates.

When evaluated in 50 trial episodes, the non-modulated method shows improvement compared to training with larger policy updates. As it avoids converging to suboptimal local maxima, the mean goal time significantly decreases to 23.76 seconds. Having adopted a more daring gait results in faster completion times, but also causes the agent to occasionally fall very early, which negatively impacts its mean final pose. Nevertheless, it maintains a higher success rate of 76%. Ultimately, the best overall performance is achieved with the PPO+CPG+mod method using smaller updates, boasting an impressive 86% success rate. While the mean goal time may not be the fastest, it compensates by demonstrating remarkable consistency, surpassing the 5-meter mark in every trial and only experiencing falls in a couple of episodes.

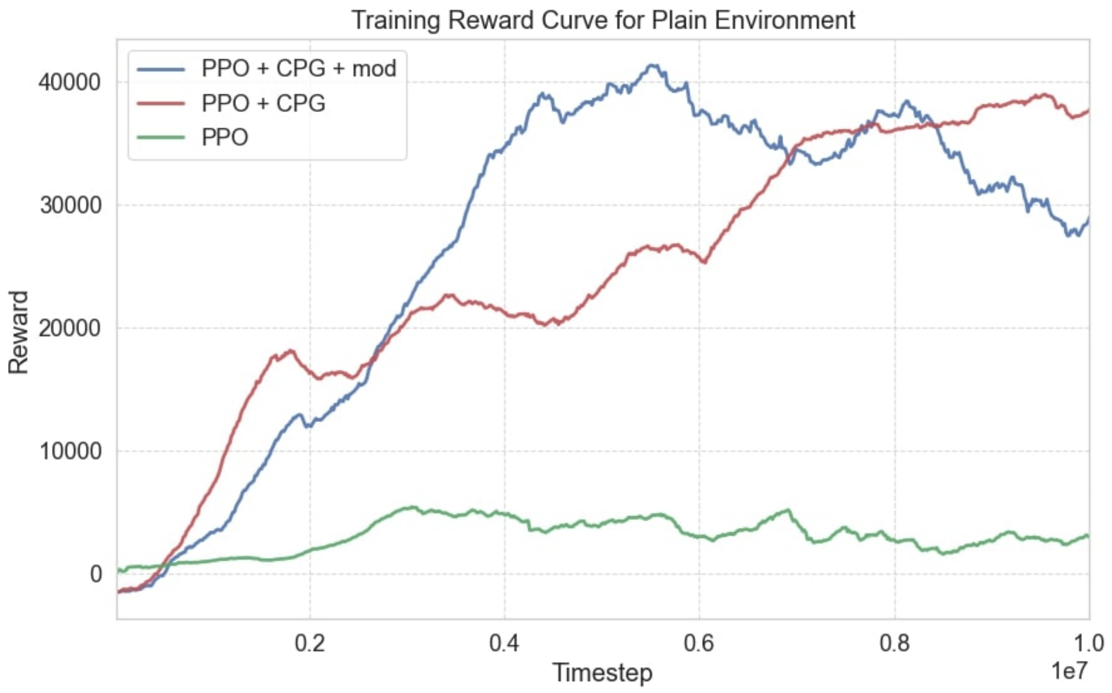


Figure 4.9: Comparison of rewards obtained during training in slope environment for three methods, using smaller policy updates

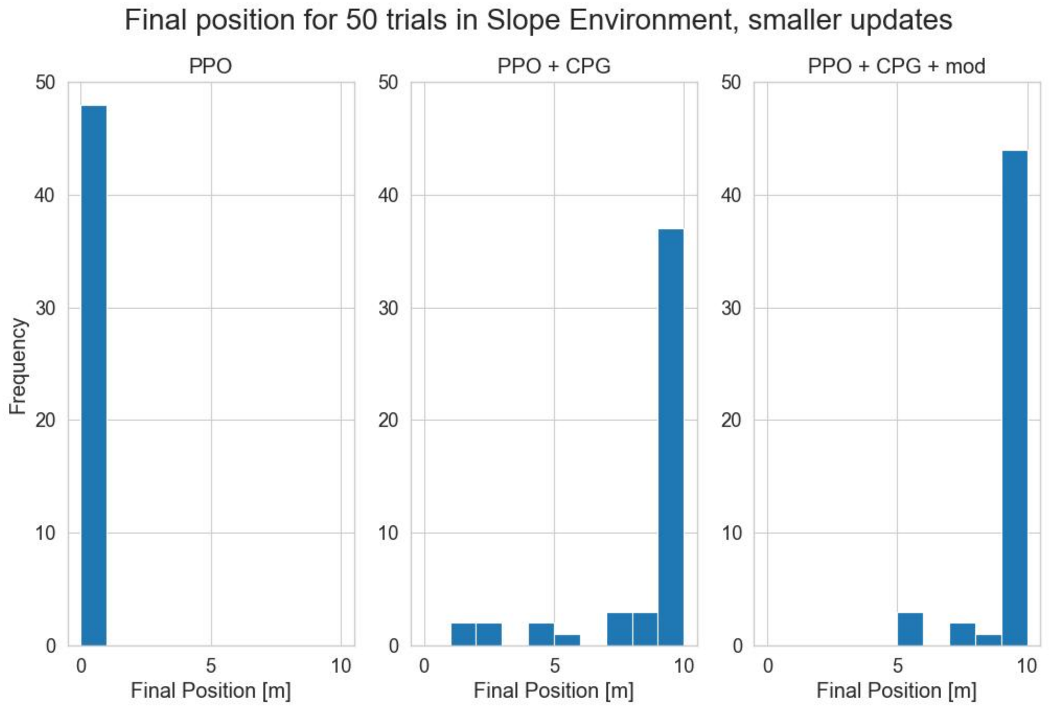


Figure 4.10: Histogram of final position reached by the agent in the slope environment for three methods, using smaller policy updates

Method	**Mean final pose [m]**	**Mean goal time [s]**	**Success rate**
PPO	0.22	N/A	0%
PPO+CPG	8.84	23.76	72%
PPO+CPG+mod	9.58	22.18	86%

Table 4.7: Performance metrics of 50 trials in slope environment with smaller policy updates

Implementing our proposed architecture, we successfully trained our quadruped robot to exhibit walking behaviors in a simulated environment, characterized by a narrow pathway with varying slopes. However, there is room for improvement in specific aspects. Although the robot can adapt its gait to transition from a horizontal surface to an upward ramp, the observed gait during the traversal of each distinct type of terrain does not exhibit significant variations. It would seem the agent learns a type of gait that works for every possible inclination, instead of learning distinct optimized gaits for each case. This doesn't imply the absence of observable differences, but rather suggests that they could be more pronounced. One potential solution is to incorporate an energy consumption metric into the reward function. By doing so, the agent would need to learn an optimal gait not only in terms of speed but also with an emphasis on energy efficiency. This approach would encourage the exploration of various motor trajectories that resemble the motions of animals. In several reviewed works, authors incorporate a maximum desired velocity v_{max} into their reward functions. Although our function currently lacks this component, its inclusion could offer several benefits. Firstly, the maximum velocity could serve as an input to dynamically adjust the robot's gait, enabling it to reach the desired speed. Secondly, utilizing an uncapped velocity reward might result in unnatural walking patterns, oscillating the motors at high frequencies, which may not be possible nor efficient on real hardware. For instance, we experimented with a broader range of possible frequencies for the CPG. However, this approach was ultimately discarded, as the policy consistently favored the highest possible frequency, leading to unrealistic robot movements through the plain environment and convergence to local maxima in the slope environment.

One limitation of this study lies in the absence of uncertainty modeling in robot state measurements. While the achieved results in a simulated environment are promising, the lack of consideration for sensor noise in proprioceptive measurements, such as motor angles and base orientation, may impact the robustness of the controller in a real-world robot platform. Enhancing the model by incorporating sensor noise modeling could contribute to a more robust and applicable control strategy in practical scenarios.

The implemented architecture is versatile, allowing for its application to various problems by replacing the VAE with a model trained on images specific to the task domain. Furthermore, given the policy's success in learning walking behaviors, transfer learning could be employed to expedite training on a new problem by reusing the previously learned primitives. We propose applying the architecture to tasks involving turning, an aspect not extensively explored in our tested environments. Potential applications include obstacle avoidance and line following. Utilizing the system for such tasks would necessitate the correction terms, as the current CPG formulation is designed for forward walking, where all legs follow the same joint trajectory with a phase shift. Furthermore, since we already posses a model for walking, this would facilitate the data collection process by automating the capture of images.

We have tested our method against the direct application of state-of-the-art reinforcement learning technique Proximal Policy Optimization (PPO), demonstrating our hypothesis that incorporating VAEs and CPGs in a hierarchical structure enhances the learning of intricate motion behaviors by improving sample efficiency. To further validate our approach, additional experiments could explore alternative state-of-the-art algorithms, such as Soft Actor Critic (SAC) and Twin Delayed DDPG (TD3). These experiments might involve substituting the RL algorithm in our architecture with the mentioned alternatives or employing them directly for motor angle selection.

Chapter 5

Conclusions

5.1. Conclusions of the Work

We introduced a method for learning adaptable walking behaviors in the reinforcement learning framework, which involves the joint integration of Variational Autoencoders for environmental feature extraction and an RL-trained policy for selecting Central Pattern Generators parameters. We demonstrated its capabilities through experiments in two distinct environments. In a simple walking-forward problem on flat terrain, our architecture proved its ability to learn to walk in under an hour of simulation time. With further training, it converged to a robust policy capable of advancing forward a distance of 10 meters in 98% of trials. In a more complex environment featuring a pathway of varying inclination, the robot also demonstrated its ability to learn to solve the task in under three hours, achieving a remarkable success rate of 86%. In both scenarios, our presented method significantly outperformed the direct application of the state-of-the-art Proximal Policy Optimization algorithm for motor angle selection.

Observing the Central Pattern Generator's parameters over time reveals that the policy generates periodic signals adapting to the current terrain the robot traverses, rather than learning fixed optimal values for each parameter. Although the difference in gaits is subtle, it becomes most noticeable during terrain transitions, where parameters vary significantly for adaptation. To achieve greater differentiation between selected gaits, the implementation of an energy consumption-based reward function is left for future work. The resulting walking motion resembles that of a quadruped animal and is learned quickly, thanks to the guidance provided by the CPG's constraints.

With the results of our experimental setups, we can declare to have demonstrated the hypothesis presented at the start of our work. We have improved the sample efficiency by integrating prior knowledge in the form of mathematical models of leg trajectories, thus enabling efficient learning of adaptable gaits capable of traversing complex environments. With this, we have also met all of our objectives of implementing and validating our proposed reinforcement learning based controller. Additionally, the relatively low training times show promise in bridging the gap between simulation and real life, but further considerations must be taken before implementation on real hardware is conducted, such as the inclusion of measurement noise which was not considered in this work.

5.2. Future Work

While our work has managed to produce great results and achieve all initially outlined objectives, there are still aspects that can be enhanced. As mentioned before, an improved reward function can be designed to better encapsulate the problem and consider new elements, such as energy efficiency to learn more realistic gaits, or a capped velocity to encourage the robot to walk at a certain speed. More thoroughly analyzing the values returned by the components of the reward function could help to better understand how it affects training and identify exploits.

The trained policies managed to reach an 86% success rate on our more complex task of walking up a path of varying slopes. While this may initially appear promising, it falls short of acceptable performance for real-world applications. Consistency and reliability are crucial in industrial use of robots, so a more robust controller capable of adapting to real-world conditions is needed. To achieve this, more realistic training environments can be developed that include more stochastic elements. Modeling sensor noise for the quadruped's measurements and adding more variation and complexity to our training grounds are some ways a more robust controller can be learnt. While policies trained in simulation might not be directly transferable to a real robot platform due to sim-to-real problems, it would give us a better idea of expected training times on hardware.

With the modular architecture of our RL controller, new Variational Autoencoder models can be trained to further improve the feature extraction, but also to adapt the controller to new tasks. For example, in an obstacle avoidance problem, our agent can learn steering behaviors to navigate while avoiding collisions and adhering to a designated path. Furthermore, transfer learning can be employed to take advantage of walking behaviors acquired from prior tasks by copying the parameter selection policy's weights, which can help decreasing training times.

Finally, additional comparisons between our system and state-of-the-art algorithms can be conducted. Moreover, we can train using other algorithms in our pipeline's reinforcement learning component, given the diverse range of available methods, with some potentially better suited to address our specific problems.

Bibliography

[1] Siegwart, R. y Nourbakhsh, I. R., Introduction to Autonomous Mobile Robots. The MIT Press, 2004.

[2] Sutton, R. S. y Barto, A. G., Reinforcement Learning: An Introduction. Cambridge, Massachusetts: The MIT Press, 2nd ed., 2018.

[3] Kumar, A., Paul, N., y Omkar, S. N., "Bipedal walking robot using deep deterministic policy gradient," CoRR, vol. abs/1807.05924, 2018, http://arxiv.org/abs/1807.05924.

[4] Pateria, S., "Hierarchical reinforcement learning: A comprehensive survey," ACM Computing Surveys, vol. 54, no. 109, pp. 1–35, 2021, doi:10.1145/3453160.

[5] Ijspeert, A. J., "Central pattern generators for locomotion control in animals and robots: A review," Neural Networks, vol. 21, no. 4, pp. 642–653, 2008, doi:https://doi.org/10.1016/j.neunet.2008.03.014. Robotics and Neuroscience.

[6] Oliehoek, F. A. y Amato, C., A Concise Introduction to Decentralized POMDPs. Springer, 1st ed., 2015.

[7] Jang, B., Kim, M., Harerimana, G., y Kim, J. W., "Q-learning algorithms: A comprehensive classification and applications," IEEE Access, vol. 7, pp. 133653–133667, 2019, doi:10.1109/ACCESS.2019.2941229.

[8] OpenAI, "Spinning Up in Deep RL," 2018, https://spinningup.openai.com/en/latest/index.html. Accessed: 27-06-2023.

[9] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., y Kavukcuoglu, K., "Asynchronous methods for deep reinforcement learning," CoRR, vol. abs/1602.01783, 2016, http://arxiv.org/abs/1602.01783.

[10] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., y Klimov, O., "Proximal policy optimization algorithms," CoRR, vol. abs/1707.06347, 2017, http://arxiv.org/abs/1707.06347.

[11] Mnih, V., Kavukcuoglu, K., Silver, D., *et al.*, "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015, http://dx.doi.org/10.1038/nature14236.

[12] Silver, D., Lever, G., Heess, N., *et al.*, "Deterministic policy gradient algorithms," en Proceedings of the 31st International Conference on Machine Learning (Xing, E. P. y Jebara, T., eds.), vol. 32 de Proceedings of Machine Learning Research, (Bejing, China), pp. 387–395, PMLR, 2014, https://proceedings.mlr.press/v32/silver14.html.

[13] Haarnoja, T., Zhou, A., Abbeel, P., y Levine, S., "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," CoRR, vol. abs/1801.01290, 2018, http://arxiv.org/abs/1801.01290.

[14] Levine, S., Kumar, A., Tucker, G., y Fu, J., "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," CoRR, vol. abs/2005.01643, 2020, https://arxiv.org/abs/2005.01643.

[15] Bellman, R. E., "The theory of dynamic programming," 1954.

[16] Arulkumaran, K., Deisenroth, M. P., Brundage, M., y Bharath, A. A., "Deep reinforcement learning: A brief survey," IEEE Signal Processing Magazine, vol. 34, no. 6, pp. 26–38, 2017, doi:10.1109/MSP.2017.2743240.

[17] Silver, D., Huang, A., Maddison, C. J., *et al.*, "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484–489, 2016, doi:10.1038/nature16961.

[18] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., y Abbeel, P., "Trust region policy optimization," CoRR, vol. abs/1502.05477, 2015, http://arxiv.org/abs/1502.05477.

[19] Sutton, R. S., Precup, D., y Singh, S., "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," Artificial Intelligence, vol. 112, no. 1-2, pp. 181–211, 1999, doi:10.1016/s0004-3702(99)00052-1.

[20] Bacon, P., Harb, J., y Precup, D., "The option-critic architecture," CoRR, vol. abs/1609.05140, 2016, http://arxiv.org/abs/1609.05140.

[21] Comanici, G. y Precup, D., "Optimal policy switching algorithms for reinforcement learning," Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 709–714, 2010.

[22] Jain, D., Iscen, A., y Caluwaerts, K., "From pixels to legs: Hierarchical learning of quadruped locomotion," CoRR, vol. abs/2011.11722, 2020, https://arxiv.org/abs/2011.11722.

[23] Badri-Spröwitz, A., Kuechler, L., Tuleu, A., Ajallooeian, M., D'Haene, M., Möckel, R., y Ijspeert, A., Oncilla robot: a light-weight bio-inspired quadruped robot for fast locomotion in rough terrain, pp. 63–64. 2011.

[24] Barasuol, V., Buchli, J., Semini, C., Frigerio, M., De Pieri, E. R., y Caldwell, D. G., "A reactive controller framework for quadrupedal locomotion on challenging terrain," en 2013 IEEE International Conference on Robotics and Automation, pp. 2554–2561, 2013, doi:10.1109/ICRA.2013.6630926.

[25] Sartoretti, G., Shaw, S., Lam, K., Fan, N., Travers, M., y Choset, H., "Central pattern generator with inertial feedback for stable locomotion and climbing in unstructured terrain," en 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 5769–5775, 2018, doi:10.1109/ICRA.2018.8461013.

[26] Tsounis, V., Alge, M., Lee, J., Farshidian, F., y Hutter, M., "Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning," CoRR, vol. abs/1909.08399, 2019, http://arxiv.org/abs/1909.08399.

[27] Xu, S., Zhu, L., y Ho, C. P., "Learning efficient and robust multi-modal quadruped locomotion: A hierarchical approach," en 2022 International Conference on Robotics and Automation (ICRA), pp. 4649–4655, 2022, doi:10.1109/ICRA46639.2022.9811640.

[28] Li, T., Lambert, N., Calandra, R., Meier, F., y Rai, A., "Learning generalizable locomotion skills with hierarchical reinforcement learning," en 2020 IEEE International

Conference on Robotics and Automation (ICRA), pp. 413–419, 2020, doi:10.1109/ICRA40945.2020.9196642.

[29] Lele, A. S., Fang, Y., Ting, J., y Raychowdhury, A., “Learning to walk: Spike based reinforcement learning for hexapod robot central pattern generation,” en 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), pp. 208–212, 2020, doi:10.1109/AICAS48895.2020.9073987.

[30] Kingma, D. P. y Welling, M., “Auto-encoding variational bayes,” 2022.

[31] Coumans, E. y Bai, Y., “Pybullet, a python module for physics simulation for games, robotics and machine learning.” http://pybullet.org, 2016–2021.

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

[33] Paszke, A., Gross, S., Massa, F., *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” en Advances in Neural Information Processing Systems 32 (Wallach, H., Larochelle, H., Beygelzimer, A., d’Alché Buc, F., Fox, E., y Garnett, R., eds.), pp. 8024–8035, Curran Associates, Inc., 2019, http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[34] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., y Dormann, N., “Stable-baselines3: Reliable reinforcement learning implementations,” Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021, http://jmlr.org/papers/v22/20-1364.html.

[35] Higgins, I., Matthey, L., Glorot, X., Pal, A., Uria, B., Blundell, C., Mohamed, S., y Lerchner, A., “Early visual concept learning with unsupervised deep learning,” 2016.

[36] Iscen, A., Caluwaerts, K., Tan, J., Zhang, T., Coumans, E., Sindhwani, V., y Vanhoucke, V., “Policies modulating trajectory generators,” CoRR, vol. abs/1910.02812, 2019, http://arxiv.org/abs/1910.02812.

[37] Gay, S., Santos-Victor, J., y Ijspeert, A., “Learning robot gait stability using neural networks as sensory feedback function for central pattern generators,” en 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 194–201, 2013, doi:10.1109/IROS.2013.6696353.