Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Generación de locomoción para un robot hexápodo usando Aprendizaje Por Refuerzo Profundo

Tesis de Licenciatura en Ciencias de la Computación

Matías Sandacz

Director: Pablo De Cristóforis[†]

Codirector: Nicolás Romero[†]

[†] Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales. Departamento de Computación. Laboratorio de Robótica y Sistemas Embebidos Buenos Aires, Argentina.

RESUMEN

En este trabajo, introducimos una metodología basada en Aprendizaje por Refuerzo para el aprendizaje de la locomoción de un robot hexápodo sobre distintos tipos de terrenos. Durante el entrenamiento, a medida que el robot va experimentando con terrenos cada vez más desafiantes, incrementaremos progresivamente su dificultad. La hipótesis central de este trabajo es que, posteriormente a la etapa de aprendizaje, el robot pueda adaptarse a terrenos no explorados durante su entrenamiento. Los experimentos realizados en simulación utilizando Gazabo muetras la factibilidad del enfoque propuesto. El sistema fue desarrollado bajo ROS (Robot Operating System) y se encuentra disponible para la comunidad en `https://github.com/lrse/hexapod-locomotion/`.

Palabras claves: Robot Hexápodo, Aprendizaje por Refuerzo, ROS, Gazebo.

Índice general

1.. Introducción 1
1.1. Motivación 1
1.2. Conceptos Preliminares 1
1.2.1. Aprendizaje por Refuerzo 1
1.2.2. Gym 15
1.2.3. Stable Baselines 16
1.2.4. Robot Operating System (ROS) 19
1.2.5. Gazebo 20

2.. Trabajos Relacionados 21

3.. Sistema Propuesto 23
3.1. Definición del Problema 23
3.1.1. Conjunto de Estados 23
3.1.2. Conjunto de Acciones 24
3.1.3. Función de Recompensa 24
3.2. Control 26
3.2.1. Esquema del robot PhantomX Mark II 26
3.2.2. Estructura General de Control 26
3.2.3. Modelo Cinemático 27
3.2.4. Control de Bajo Nivel 28
3.3. Generación de Terrenos 29
3.3.1. Ruido de Perlín 29
3.4. Entrenamiento con Currícula de Terrenos 33
3.4.1. Generación Aleatoria de Terrenos 33
3.4.2. Generación Adaptativa de Terrenos 34
3.4.3. Deseabilidad 34
3.4.4. Variables de Medida 35
3.5. Detalles de Implementación 36
3.5.1. Phantomx.py 37
3.5.2. Server.cpp 37
3.5.3. Gait-Controller.cpp 38
3.5.4. Listado de topicos 38
3.5.5. Plugins de Gazebo 40

4.. Resultados 42

5.. Conclusiones y Trabajo Futuro 47

1. INTRODUCCIÓN

1.1. Motivación

Los robots con patas articuladas se destacan por su capacidad para realizar tareas en entornos desafiantes, hostiles o inaccesibles para otros robots terrestres con sistemas de locomoción basado en ruedas u orugas, al mismo tiempo que pueden transportar más carga y operar durante más tiempo que los vehículos aéreos no tripulados. Sus habilidades para realizar movimientos omnidireccionales y atravesar terrenos irregulares los convierten en una excelente opción para diversas aplicaciones, como las misiones de Búsqueda y Rescate Urbano (USAR por sus siglas en inglés) en construcciones colapsadas en situaciones de catástrofe, para operar en ambientes subterráneos como minas e incluso para la exploración extra planetaria. Sin embargo, es sabido que el control de movimiento de este tipo de robots es sumamente desafiante debido a la cantidad de grados de libertad. Por otro lado, el control clásico requiere conocer en detalle la física del robot lo cual agrega una dificultad extra y limita su portabilidad a otros sistemas. El enfoque de este trabajo se basa en explorar una alternativa a las estrategias de control tradicional que es el uso de aprendizaje por refuerzo (RL del inglés Reinforcement Learning). Estos algoritmos permiten a un robot con patas articuladas aprender a transitar terrenos desafiantes de manera autónoma. Esta capacidad de adaptación es fundamental para poder enfrentar terrenos irregulares y cambiantes, proporcionando una solución robusta para la exploración de ambientes desconocidos.

1.2. Conceptos Preliminares

1.2.1. Aprendizaje por Refuerzo

Interacción entre el Agente y el Ambiente

En todo problema de aprendizaje por refuerzo contamos con un agente y un ambiente que interactúan continuamente. El agente se encuentra en algún estado dentro del ambiente, y selecciona una posible acción. En consecuencia, el ambiente responde a dichas acciones posicionando al agente en un nuevo estado. El ambiente también genera una recompensa que el agente intentará maximizar a lo largo del tiempo.

Dicha interacción se da en una secuencia de pasos discretos en el tiempo $t = 0, 1, 2, 3....$ En cada paso de tiempo t, el agente recibe una representación del estado $S_t \in S$ y selecciona una acción $a \in A$, donde A es el conjunto de todas las posibles acciones. Un paso de tiempo más tarde, el agente recibe una recompensa numérica $R_{t+1} \in R$ y transiciona a un nuevo estado $S_{t+1} \in S$.

En cada paso del tiempo t, el agente utiliza una regla para seleccionar la siguiente acción. Esta regla se conoce como la política del agente, y se puede interpretar como el comportamiento del agente.

En esencia, los métodos de aprendizaje por refuerzo especifican como hace un agente para modificar su política (comportamiento) a medida que observa distintos flujos de

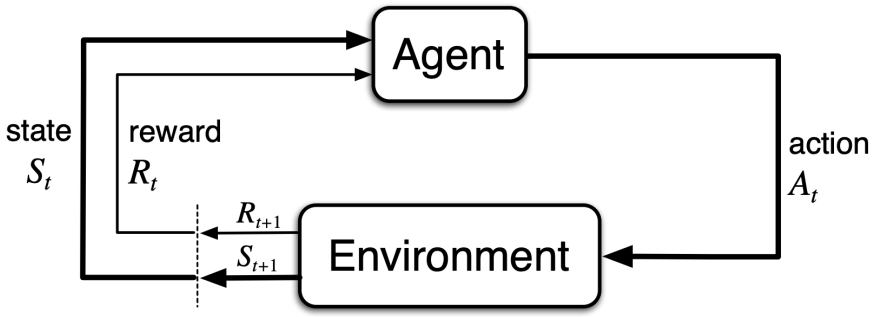

Fig. 1.1: La interacción entre un agente y el ambiente.

experiencias, con el objetivo de que sus acciones maximicen la recompensa obtenida a lo largo del tiempo.

Políticas

Como mencionamos anteriormente, una política es una regla utilizada por el agente para decidir que acción tomar. Puede ser una función determinística, usualmente denotada

$$a_t = \mu(s_t) \tag{1.1}$$

o puede ser estocástica. En este caso, la política será una distribución de probabilidad condicional sobre las acciones, y la denotamos:

$$p(a_t|s_t). \tag{1.2}$$

En el área de Aprendizaje por Refuerzo Profundo, solemos utilizar políticas parametrizadas, es decir, funciones cuya salida depende de un conjunto de parámetros que podemos optimizar mediante algún algoritmo de optimización. Se suele llamar θ a los parámetros de la política, y se denota:

$$a_t = \mu_\theta(s_t) \tag{1.3}$$

$$a_t \sim \pi_\theta(.|s_t) \tag{1.4}$$

La diferencia entre el Aprendizaje por Refuerzo y el Aprendizaje por Refuerzo Profundo, es que en este último se utilizan redes neuronales profundas para representar los distintos componentes, como por ejemplo la política del agente. Las redes neuronales suelen ser más dificil de optimizar que otro tipo de funciones, como por ejemplo un polinomio. Sin embargo, tienen propiedades muy deseables como ser aproximadores universales de funciones [1].

Políticas Determinísticas

Un ejemplo de una política determinística, es una red neuronal con una única capa oculta, como se puede observar en la figura 1.2.

En esta arquitectura concreta, la entrada es un único estado representado como un vector en i dimensiones, y la salida es una única acción representada como un vector en

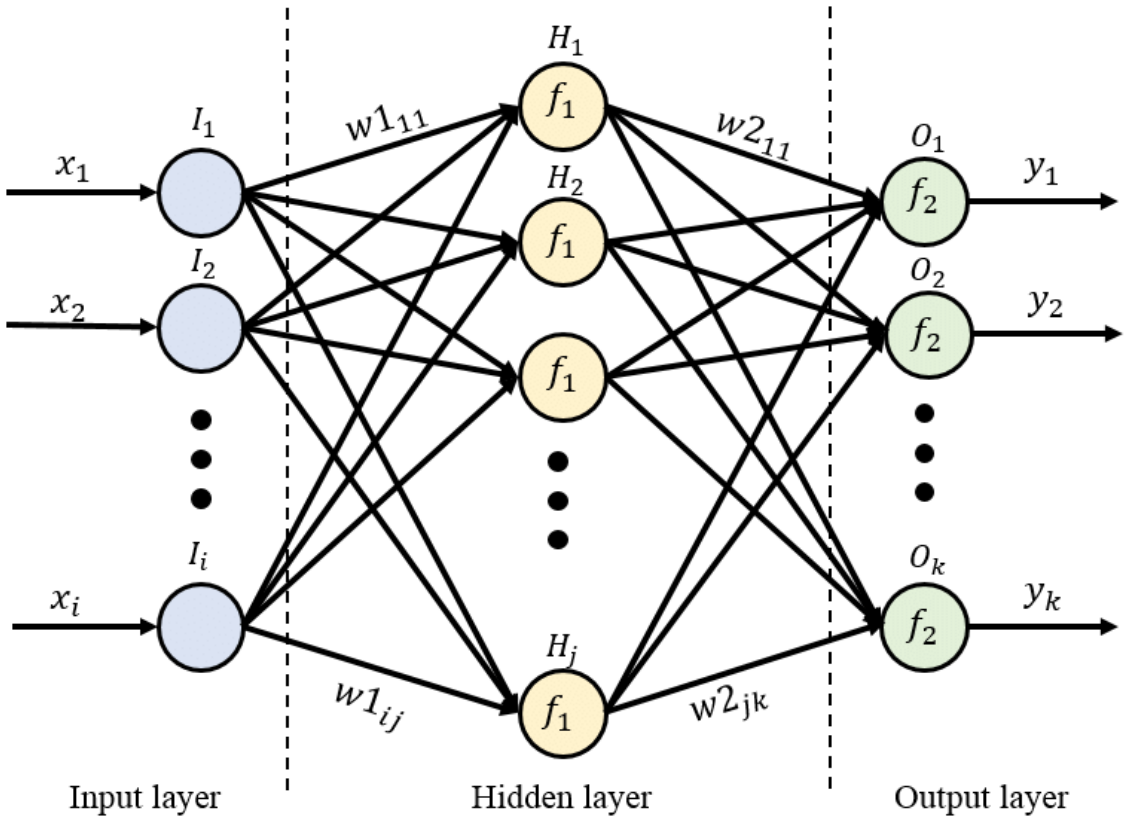

Fig. 1.2: Política deterministica representada como una red neuronal con i entradas, una única capa oculta, y k salidas.

k dimensiones. Es importante observar que la salida es una única acción, y no una distribución, ya que estamos en el caso determinístico. Muchas veces no es posible representar un estado o una acción en una única dimensión, por ende es necesario utilizar un vector de más dimensiones.

Políticas Estocásticas

Los dos tipos más comunes de políticas estocásticas utilizadas en Aprendizaje por Refuerzo Profundo son las políticas categóricas, y las políticas Gaussianas.

Las políticas categóricas se suelen utilizar en espacios de acciones discretos, mientras que las políticas Gaussianas se suelen utilizar en espacios de acciones continuos. En este trabajo, estudiaremos el caso continuo.

Tal como su nombre indica, una política Gaussiana utiliza una distribución Gaussiana para modelar una distribución de probabilidad condicional sobre las acciones.

Una distribución Gaussiana multivariada (o distribución normal multivariada), se describe con un vector de medias μ, y una matriz de covarianza Σ. En el caso especial en donde la matriz es diagonal, la matriz puede ser representada por un vector σ_θ.

Una política Gaussiana diagonal puede ser representada como una red neuronal que transforma un estado en un vector de medias para las acciones $\mu_\theta(s)$. Los desvíos σ_θ se definen como hiperparámetros.

Dado el vector de medias $\mu_\theta(s)$, un vector de desviación estándar σ_θ (que en general no es función del estado), y un ruido normal $z \sim \mathcal{N}(0, I)$, podemos realizar el muestreo de una acción de la siguiente manera:

$$a = \mu_\theta(s) + (\sigma_\theta(s) \odot z),$$

donde $\odot$ se define como el producto componente a componente entre dos vectores.

Objetivos y Recompensas

En los problemas de Aprendizaje por Refuerzo, la noción de un objetivo o tarea a cumplir se formaliza en términos de una señal llamada recompensa. En cada paso del tiempo t, la recompensa es simplemente un número real $r_t \in \mathbb{R}$. A grandes rasgos, el objetivo del agente es maximizar la cantidad de recompensa obtenida. Esto no solamente significa maximizar la recompensa inmediata, sino que la recompensa acumulada a largo plazo. Podemos expresar esta noción informal que acabamos de describir utilizando lo que se conoce como la Hipótesis de la Recompensa:

Las tareas pueden pensarse como la maximización del valor esperado de la suma acumulada de la recompensa. [2].

El uso de una señal de recompensa para formalizar la idea de un objetivo es uno de los aspectos más distintivos del Aprendizaje por Refuerzo.

A primera vista, esto podría parecer limitante. ¿Realmente todo objetivo se puede pensar como la maximización de una señal? Sin embargo, hay una gran cantidad de ejemplos que muestran la flexibilidad de esta hipótesis. Por ejemplo, para que un robot aprenda a caminar, podemos otorgarle una recompensa proporcional a su desplazamiento. En cambio, si queremos que un robot aprenda a escapar de un laberinto, podríamos proporcionar una recompensa negativa por cada unidad de tiempo hasta que logre escapar; esto incentiva al robot a escapar lo más rápido posible.

En esencia, debemos establecer una recompensa de forma tal que maximizarla implique alcanzar nuestros objetivos.

Trayectorias

En algunos problemas podemos dividir la interacción entre el agente y el ambiente en secuencias llamadas trayectorias. Las trayectorias a veces también se suelen llamar episodios o rollouts.

Formalmente, una trayectoria τ es una secuencia de estados y acciones dentro del ambiente.

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

El primer estado dentro del ambiente, s_0, se elige aleatoriamente de una distribución inicial, denotada ρ_0:

$$s_0 \sim \rho_0(.)$$

Dado un estado s_t y una acción a_t, el ambiente transiciona a un nuevo estado s_{t+1}

y otorga una recompensa r_{t+1} de acuerdo a las leyes de su dinámica, que en general son desconocidas para nosotros.

$s_{t+1}, r_{t+1} \sim \mathcal{P}[.|s_t, a_t]$

Como la dinámica del sistema únicamente depende del estado y la acción en el tiempo inmediatamente anterior, diremos que se cumple la Propiedad de Markov.

Propiedad de Markov

En un problema de Aprendizaje por Refuerzo, un conjunto de estados es Markoviano si y solo si cumple con la propiedad de Markov [3, p. 76]:

$\mathcal{P}(S_{t+1} = s', R_{t+1} = r|S_0, A_0, R_0, ..., S_t, A_t, R_t) = \mathcal{P}(S_{t+1} = s', R_{t+1} = r|S_t, A_t).$

Esto quiere decir que el estado S_t es suficiente por sí solo para definir la dinámica del sistema, y podemos descartar el resto del historial de experiencias.

Función de Retorno

Hasta ahora discutimos el objetivo del aprendizaje por refuerzo informalmente, y mencionamos que al agente le interesa maximizar la suma acumulada de la recompensa a largo plazo. Supongamos que la secuencia de recompensas que el agente recibe luego del tiempo t se define como:

$R(s_t, a_t, s_{t+1}) = r_t.$

Dada una trayectoria τ y un tiempo t, podemos definir el retorno desde t como:

$G_t = R_0 + R_1 + ...R_T$, donde T es el tiempo final del episodio.

Sin embargo, en la gran mayoría de los casos se utiliza el retorno descontado [3, p. 84]:

$G_t = \sum_{k=0}^{\infty} {}^{k} R_{t+1+k}$, en donde $0 \leq\leq 1$ es el *factor de descuento.*

El factor de descuento determina la relevancia de las recompensas a medida que avanzamos en el tiempo. En el caso donde $= 0$, al agente únicamente le va a importar la recompensa inmediata R_{t+1}. A medida que $\to 1$, se le da más importancia a las recompensas futuras.

Finalmente, podemos definir el retorno de una trayectoria τ como $R(\tau) = G(0)$.

El Problema del Aprendizaje por Refuerzo

El problema de aprendizaje por refuerzo consiste en encontrar una política que maximice $J(\pi) = E_{\tau\sim\pi}[R(\tau)]$.

Para poder hablar sobre $E_{\tau\sim\pi}[R(\tau)]$, primero debemos mencionar la distribución de probabilidad sobre las distintas trayectorias. Para esto, supongamos una política estocástica π fija, y τ una trayectoria cualquiera:

Como consecuencia de aplicar la propiedad de Markov, podemos calcular la probabilidad de seguir una trayectoria τ bajo la política π:

$\mathcal{P}(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} \mathcal{P}(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$.

Luego, utilizando directamente la definición matemática, el valor esperado del retorno es:

$E_{\tau\sim\pi}[G(\tau)] = \int_\tau \mathcal{P}(\tau|\pi)R(\tau)$

El problema central de optimización en Aprendizaje por Refuerzo puede ser expresado de la siguiente manera:

$\pi_*(a|s) = \arg\max_\pi E_{\tau\sim\pi}[R(\tau)]$

donde llamamos a π_* una política óptima.

Proceso de Decisión de Markov

Los Procesos de Decisión de Markov son la herramienta principal que vamos a utilizar para formalizar la noción de una instancia de un problema de Aprendizaje por Refuerzo. Formalmente, un Proceso de Decisión de Markov, es una cinco-upla $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, >$, tal que:

- $\mathcal{S}$ es un conjunto finito de estados que satisface la propiedad de Markov.
- $\mathcal{A}$ es un conjunto finito de acciones.
- $\mathcal{P}$ representa la dinámica del sistema.
- $\mathcal{R}$ es la función de recompensa.
- γ tal que $0 \leq \gamma \leq 1$ es el factor de descuento.

Ejemplo:

En esta sección veremos un ejemplo concreto de un Proceso de Decisión de Markov, y que herramientas podemos usar para visualizarlo.

Supongamos que un robot móvil tiene la tarea de recoger latas vacías en una oficina. El robot cuenta con varios sensores para detectar objetos, y un brazo y pinza que puede usar para recogerlas y colocarlas en un contenedor de basura. El sistema de control del robot tiene componentes para interpretar la información sensorial, para navegar y para controlar el brazo y la pinza. Las decisiones de alto nivel sobre cómo buscar latas las toma un agente de aprendizaje por refuerzo basado en el nivel de carga actual de la batería.

Para dar un ejemplo sencillo e ilustrativo, suponemos que en nuestro problema sólo se pueden distinguir dos niveles de carga de la batería. Esto resulta en un pequeño conjunto de estados de dos elementos $\mathcal{S} = \{alto, bajo\}$.

En cada estado, el agente puede decidir si (1) activamente buscar una lata durante un cierto período de tiempo, (2) permanecer quieto y esperar que alguien le traiga una lata, o (3) regresar a su base de operaciones para recargar su batería.

Cuando el nivel de energía es alto, no tiene sentido recargar la batería, por eso no lo incluimos en las posibles acciones asociadas a ese estado. Los conjuntos de acciones son entonces $\mathcal{A}(alto) = \{buscar, esperar\}$, $\mathcal{A}(bajo) = \{buscar, esperar, recargar\}$.

Las recompensas son cero la mayor parte del tiempo, pero se vuelven positivas cuando el robot consigue una lata vacía. En cambio, la recompensa será negativa si la batería se agota por completo.

La mejor manera de encontrar latas es buscarlas activamente, pero esto agota la batería del robot, mientras que esperar no. Siempre que el robot está buscando, existe la posibilidad de que su batería se agote. En este caso el robot debe apagarse y esperar a ser rescatado (produciendo una recompensa negativa). Si el nivel de energía es alto, entonces se puede iniciar un período de búsqueda activa sin riesgo de agotar la batería.

Un período de búsqueda que comienza con un nivel de energía alto deja el nivel de energía alto con probabilidad α y la reduce a un nivel bajo con probabilidad $1 - \alpha$.

Por otra parte, si se inició un período de búsqueda cuando el nivel de energía es bajo, lo deja bajo con probabilidad β y agota la batería con probabilidad $1 - \beta$. En este último caso, es necesario rescatar el robot y recargar la batería.

Cada vez que el robot logre recoger una lata, tendrá una recompensa unitaria, mientras que una recompensa de -3 cada vez que haya que rescatar al robot.

Sean r_{search} y r_{wait}, tal que $r_{search} > r_{wait}$ el número esperado de latas que el robot recogerá (y por tanto la recompensa esperada) mientras busca y espera respectivamente.

Este problema se puede representar como un proceso de Decisión de Markov finito, y podemos utilizar una tabla o un grafo para describir la dinámica del sistema, especificados en 1.3 y 1.4.

s	a	s'	$p(s' \mid s, a)$	$r(s, a, s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1 - \alpha$	$r_{\texttt{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-

Fig. 1.3: Tabla de probabilidades de transición y recompensas para el Proceso de Decisión de Markov. Cada fila representa una posible combinación de estado a tiempo t, acción, y estado a tiempo $t + 1$ [3, p. 83].

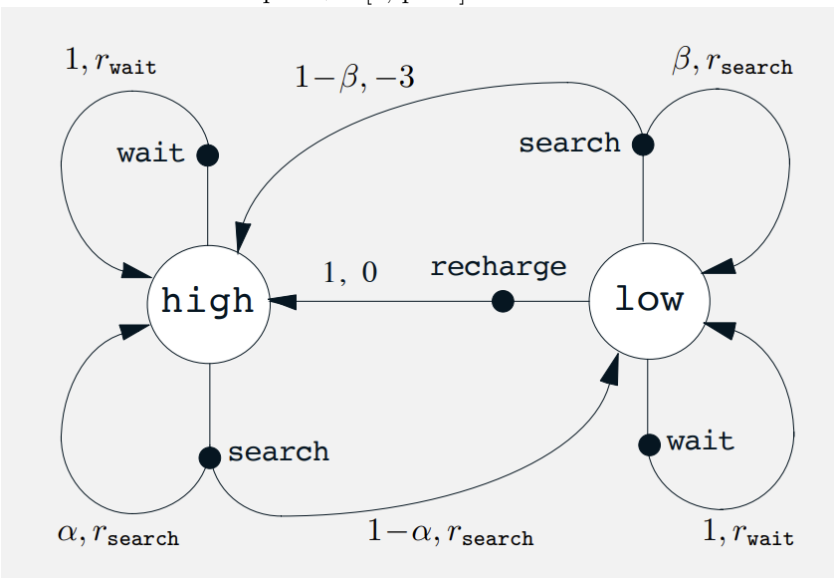

Fig. 1.4: Grafo de transición para el Proceso de Decisión de Markov. Los nodos representan estados y los arcos contienen información acerca de la acción, probabilidad de transición al siguiente estado y la recompensa [3, p. 83].

Función de Valor

En la gran mayoría de los problemas de Aprendizaje por Refuerzo se busca estimar funciones de valor - funciones cuyo dominio son los posibles estados, y estiman cuan bueno es estar en ese estado para el agente. El valor de un estado, bajo la política π, denotado:

$$v_\pi(s),$$

se define como el retorno esperado al comenzar en el estado s y luego seguir la política π. Formalmente,

$$v_\pi(s) = E_\pi[R(\tau)|S_t = s].$$

Función Q

De manera similar, podemos extender el dominio de la función de valor para que tome un par (estado,acción). Definimos el valor de tomar la acción a en el estado s bajo la política π como:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a].$$

Funciones de Valor óptimas

Clasificación de algoritmos en Aprendizaje por Refuerzo (AR)

Luego de haber repasado los conceptos básicos y la terminología utilizada en los problemas de Aprendizaje por Refuerzo, podemos explorar el panorama de los distintos algoritmos modernos, junto con las ventajas y las desventajas de cada uno.

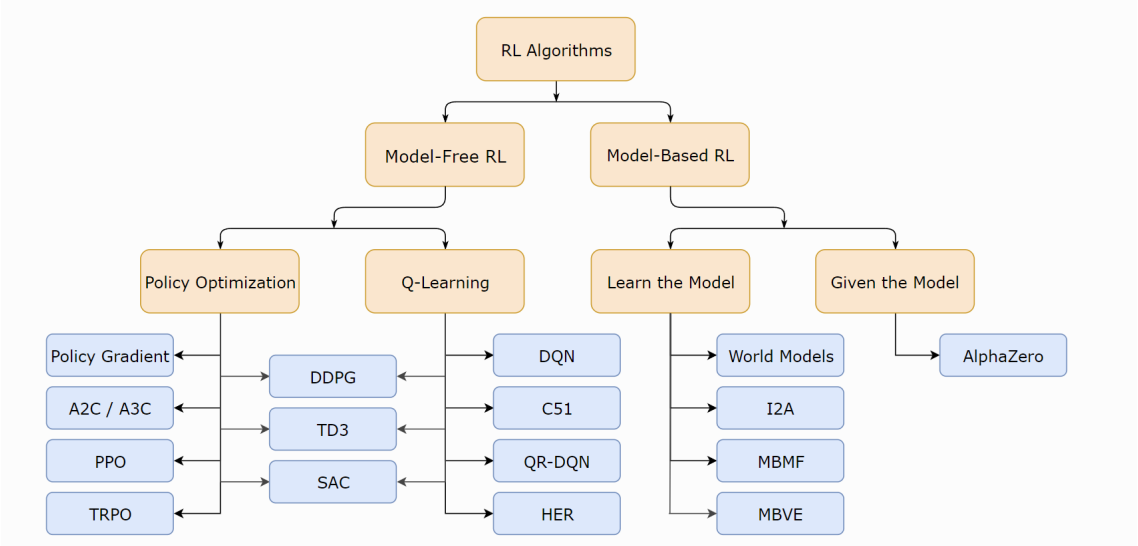

Fig. 1.5: Una categorización no exhaustiva de los distintos algoritmos de Aprendizaje por Refuerzo [4]

AR basado y libre de modelos

Uno de los puntos más importantes al caracterizar un algoritmo de AR, es si el agente conoce, o aprende, el modelo que define la dinámica del sistema. Cuando hablamos de la dinámica del sistema, nos referimos a la función de probabilidad $s_{t+1}, r_{t+1} \sim \mathcal{P}[.|s_t, a_t]$ utilizada por el ambiente para transicionar al siguiente estado y otorgar recompensas.

Una de las principales ventajas de AR basado en modelos es que el agente puede aprender una política óptima utilizando menos datos. El modelo le otorga al agente conocimiento previo que puede ser explotado [5]. Un ejemplo famoso de este enfoque es AlphaZero [6].

Sin embargo, en la gran mayoría de los casos el agente no tiene acceso al modelo $\mathcal{P}$. Si un agente quiere usarlo, deberá aprenderlo. El aprendizaje de modelos es un problema fundamentalmente díficil, por lo cual no es muy utilizado en la práctica.

Los algoritmos que utilizan un modelo se denominan métodos basados en modelos, y los que no lo utilizan se denominan métodos libres de modelos. En general, los modelos libres de modelos son más fáciles de implementar y entrenar, y son populares en la práctica. En este trabajo, nos centraremos en este tipo de modelos.

¿Qué aprender en AR libre de modelos?

Otro de los puntos clave al caracterizar un algoritmo es decidir qué componente queremos aprender. Habitualmente, se suele elegir entre los siguientes componentes:

- Políticas - ya sean estocásticas o determinísticas
- Funciones de valor $v_\pi(s)$
- Función de valor estado-acción $q_\pi(s, a)$

Q-Learning

Los métodos de esta familia aprenden un aproximador $q_\pi(s, a)$ para la función de valor acción óptima, $q^*_\pi(s, a)$. La política correspondiente se obtiene maximizando el aproximador:

$$\pi(a|s) = \arg\max_a q_\pi(s, a).$$

Es importante notar que esta familia de métodos requieren que el espacio de acciones sea finito, para poder computar la maximización sobre las acciones. En el presente trabajo, contamos con un espacio de acción infinito, por ende debimos descartar dicha familia de métodos.

Optimización de la Política

Los métodos de esta familia representan una política parametrizada explícitamente como $\pi_\theta(a \mid s)$ y optimizan los parámetros θ mediante algun método, como ascenso de gradiente, utilizando como función objetivo el valor esperado del retorno, $J(\pi_\theta)$, que estudiamos en las secciones previas.

La optimización de políticas también suele implicar aprender un aproximador $V_\phi(s)$ para la función de valor de la política $V^\pi(s)$, que se utiliza para guiar la actualización de los parámetros θ. Algunos ejemplos de algoritmos que pertenecen esta familia son:

- Política básica basada en gradientes: Es el más sencillo dentro de esta familia. Da pasos pequeños en cada actualización.
- PPO: Es el algoritmo principal en el cual basamos nuestro trabajo y estudiaremos en detalle. Este algoritmo maximiza $J(\pi_\theta)$ indirectamente, utilizando una función objetivo que controla la velocidad de actualización de los parámetros θ.
- TRPO: Similar a PPO pero utiliza la divergencia de Kullback-Leibler para controlar la velocidad de actualización de los parámetros.

La principal ventaja de estos métodos, es que aprenden directamente lo que nos interesa: la política. Debido a esto, suelen ser más estables y confiables que otros métodos, como por ejemplo Q-learning, en donde la política se aprende indirectamente maximizando la función de valor. Otro punto muy importante, es que al optimizar directamente los parámetros θ, nos permiten utilizar espacios de acciones continuos. [4] En este trabajo nos centraremos en estudiar estos tipos de métodos.

Política básica basada en gradientes

En esta sección vamos a introducir los métodos de optimización de la política. Consideramos el caso de una política estocástica parametrizada π_θ. Nuestro objetivo es maximizar la recompensa esperada.

$$J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}}[R(\tau)].$$

Queremos optimizar la función de costo J mediante el método de ascenso de gradiente, utilizando una velocidad de aprendizaje dada por α

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

El gradiente $\nabla_\theta J(\pi_\theta)$ se denomina gradiente de política, y los algoritmos que optimizan una política de esta manera se denominan algoritmos de optimización de la política.

Se puede realizar una derivación matemática de $J(\pi_\theta)$ y luego utilizando la Ley de los Grandes Números se obtiene una estimación de la expresión matemática que queremos maximizar. La demostración completa se puede ver en [4].

Según [4], si recopilamos un conjunto de trayectorias $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$, donde cada trayectoria se obtiene dejando que el agente actúe en el entorno utilizando la política π_θ, el gradiente de la política se puede estimar como

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(a_t \mid s_t)\,R(\tau),$$

donde $|\mathcal{D}|$ es el número de trayectorias en $\mathcal{D}$.

Esta es la versión más simple de la expresión que deseamos optimizar. Simplemente ejecutamos la política en un entorno para recopilar un conjunto de datos de distintas trayectorias, calculamos el gradiente de la política, y actualizamos los parámetros con un paso de ascenso de gradiente.

Uso de un Baseline Podemos reescribir la función objetivo utilizando una función Φ_t llamada Baseline. Se puede demostrar que se mantiene la esperanza matemática, reduciendo significativamente la varianza [4]. Esto incrementa la velocidad del entrenamiento.

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) \Phi_t,$$

en general Φ_t se define como la función de ventaja:

$$A^\pi \left(s_t, a_t\right) = q_\pi \left(s_t, a_t\right) - v_\pi \left(s_t\right)$$

El baseline $A^\pi \left(s_t, a_t\right)$ describe la ventaja de tomar una acción a en comparación a la política actual. Como no conocemos a la función de valor v_π, la podemos estimar minimizando el Error Cuadrático Medio.

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| \, T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(v_\phi \left(s_t\right) - \hat{R}_t\right)^2$$

TRPO y PPO

Uno de los problemas con Proximal Policy Optimization (PPO) y Trust Region Policy Optimization (TRPO), es que políticas cercanas en el espacio de parámetros pueden llevar a enormes diferencias en las trayectorias generadas. Un solo paso de actualización de los parámetros equivocado puede arruinar completamente el entrenamiento, ya que las trayectorias generadas con la nueva política pueden no ser útiles para el aprendizaje. Los metodos de PPO y TRPO evitan este tipo de colapso, y rápida y monótonamente mejoran la performance.

La gran diferencia entre estos métodos y el método básico presentado anteriormente de políticas basadas en gradientes, es que se actualizan los parámetros θ de acorde al parámetro de aprendizaje α que suele ser muy pequeño. Esto lleva a un aprendizaje lento.

Los métodos TRPO y PPO actualizan la política satisfaciendo restricciones especiales sobre la máxima distancia que se permite entre la política anterior y la actual.

En TRPO, esta restricción se expresa en términos de divergencia Kullback-Leibler (KL), que es una medida para cuantificar la diferencia entre dos distribuciones de probabilidad, resultando en un problema de optimización restringida. [4]

Por otro lado, PPO utiliza métodos de primer orden junto con manipulaciones matemáticas para mantener las políticas cercanas entre si. PPO es significativamente más sencillo de implementar, y empíricamente es al menos tan performante como TRPO. Es por esta razón que lo elegimos para nuestro trabajo.

PPO

PPO apunta a aumentar la probabilidad de las acciones que nos dan mayor recompensa hasta converger a la política óptima de forma controlada. El algoritmo actualiza los pesos maximizando la siguiente función objetivo, que ahora depende de los parámetros anteriores.

$$\theta_{k+1} = \arg\max_\theta \mathop{\mathrm{E}}_{s,a \sim \pi_{\theta_k}} \left[L \left(s, a, \theta_k, \theta\right)\right]$$

La maximización se realiza comunmente utilizando múltiples pasos de Ascenso de Gradiente Estocástico (SGD).

En dicha expresión, L se define como:

$$L\left(s, a, \theta_k, \theta\right) = \text{mín}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \quad g\left(\epsilon, A^{\pi_{\theta_k}}(s, a)\right)\right),$$

donde

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

y ϵ es un hiperparámetro (pequeño) que indica la máxima distancia permitida entre la nueva política y la vieja.

Para ganar un poco más de intuición de cómo esta expresión puede controlar la distancia entre dos políticas consecutivas, consideremos un par estado-acción concreto (s, a) y pensemos en el caso en donde la función de ventaja A es positiva para ese par estado-acción.

En ese caso, la contribución a la función objetivo de ese par (s, a) está dado por:

$$L\left(s, a, \theta_k, \theta\right) = \text{mín}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, (1+\epsilon)\right) A^{\pi_{\theta_k}}(s, a)$$

Como la función de ventaja es positiva, la función objetivo va a incrementar siempre y cuando las probabilidades de la acción a sean mayores a las probabilidades de la política vieja, es decir si $\pi_\theta(a \mid s)$ aumenta.

Sin embargo, el término de minimización pone un límite sobre cuan rápido la función objetivo puede incrementar. Una vez que $\pi_\theta(a \mid s) > (1+\epsilon)\pi_{\theta_k}(a \mid s)$, el mínimo entra en juego y este término tiene un límite de $(1+\epsilon)A^{\pi_{\theta_k}}(s, a)$. Por ende, la nueva política no tiene incentivos de alejarse demasiado de la política vieja.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Fig. 1.6: Pseudocódigo del algoritmo PPO-Clip. [4]

1.2.2. Gym

Gym es una biblioteca de código abierto implementada en Python para desarrollar entornos de aprendizaje por refuerzo al proporcionar una API estándar común para todos los entornos. Cualquier entorno de aprendizaje por refuerzo debe poder realizar las siguientes operaciones:

- *Step:* Actualiza el entorno con una acción que devuelve la siguiente observación del agente, junto con la recompensa por realizar esa acción. También indica si el entorno finaliza o se trunca debido a las últimas acciones e información del entorno sobre ese paso, es decir, información adicional que el usuario quiera agregar.
- *Reset:* Es responsable de resetear el entorno al estado inicial. Es requerido antes del primer llamado a Step.

El entorno tiene también algunas propiedades adicionales

- *Action Space:* Un conjunto de acciones válidas dentro del cual deben estar contenidas todas las acciones válidas. Define el rango de valores para cada una de las acciones.
- *Observation Space:* Un conjunto de observaciones válidas dentro del cual deben estar contenidas todas las observaciones válidas. Define el rango de valores para cada una de las observaciones.
- *Reward Range:* El rango de valores para las recompensas.

La interfaz *gym.Env* que deben implementar los entornos, incluido el nuestro para definir al comportamiento del robot hexápodo, tienen la siguiente estructura:

```
class Env:

    def step(self, action) -> tuple[ObsType,
                                    SupportsFloat,
                                    bool,
                                    bool,
                                    dict[str, Any]]

    def reset(self) -> tuple[ObsType, dict[str, Any]]
```

El **step** toma como parámetro la acción proporcionada por el agente para actualizar el estado del entorno, y devuelve:

- Observación (ObsType): un elemento del espacio de observación del entorno que sirve como la siguiente observación para la operación del agente.
- Recompensas (SupportsFloat): Recompensas por realizar la acción.
- Terminación (bool): si el agente alcanza el estado de terminación.
- truncado (bool): si se cumple la condición de truncamiento. Normalmente se trata de un límite de tiempo.

- info (dict): contiene información de diagnóstico auxiliar (útil para depurar, aprender y registrar).

El **reset** debe llamarse antes de comenzar un nuevo episodio, o cuando el step retorna que el episodio ha sido truncado o finalizado. Este método genera un nuevo estado inicial con un cierto grado de aleatoriedad para garantizar que el agente pueda explorar todo el espacio de estados.

Como veremos más adelante, en el presente trabajo implementamos el entorno del Robot Hexápodo como una clase que implementa la interfaz *Gym.env*. Esto nos permite utilizar bibliotecas tales como Stable Baselines para el entrenamiento.

1.2.3. Stable Baselines

Stable Baselines es una biblioteca ampliamente utilizada para entrenar algoritmos de aprendizaje por refuerzo sobre distintos tipos de entornos que implementen la interfaz de Gym vista anteriormente.

La ventaja de utilizar una biblioteca de este tipo es evitar programar algoritmos de aprendizaje por refuerzo desde cero, reduciendo tiempos de desarrollo, implementación y depuración de código. De esta forma, utilizamos implementaciones proporcionadas por bibliotecas que han sido ampliamente probadas, por lo que podemos centrarnos en definir el agente y su interacción con el entorno.

A continuación, podemos ver un fragmento de código que muestra el esqueleto que normalmente utiliza una aplicación de aprendizaje por refuerzo escrita utilizando Stable Baselines.

```
import gym

from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines import PPO

env = gym.make('CartPole-v1')

model = PPO(MlpPolicy, env, verbose=1)
model.learn(total_timesteps=10000)
```

En este ejemplo estamos utilizando las bibliotecas Gym y Stable Baselines. Utilizamos Gym para inicializar un entorno para el famoso problema CartPole, conocido también como péndulo invertido, descrito por Barto, Sutton y Anderson en el libro "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem". En este problema, una varilla está conectada a un carro mediante juntas rígidas y el carro se mueve a lo largo de una pista sin fricción. El péndulo se coloca verticalmente sobre el carro con el propósito de equilibrar la varilla ejerciendo fuerza sobre el carro en las direcciones izquierda y derecha. Usamos la biblioteca gym, lo que nos proporciona diferentes tipos de entornos conocidos.

Por otro lado, le decimos a Stable Baselines que queremos usar el algoritmo de aprendizaje PPO para entrenar a nuestro agente y que este debe ser controlado por una red neuronal tipo Perceptrón Multi Capa (MLP) clásica. Stable baselines nos permite definir cualquier tipo de arquitectura para la red neuronal, pero por defecto la MLP consiste de dos capas de 64 neuronas.

Finalmente, llamamos a la función de aprendizaje para iniciar el proceso de entrenamiento e indicar el número de cada iteración en la que queremos actualizar los pesos de la red neuronal.

Callbacks

La función de entrenamiento puede recibir como argumentos otras funciones llamadas callbacks, que serán llamadas dentro del agente. Esto es útil si queremos mostrar curvas de aprendizaje en tiempo real en Tensorboard o guardar los pesos de agentes periódicamente. TensorBoard es un conjunto de herramientas de visualización, que sirve para depurar, optimizar y entender los resultados de los algoritmos de aprendizaje automático.

```
model.learn(
    total_timesteps=4000000,
    progress_bar=True,
    callback=[checkpoint_callback, TensorBoardCallback()]
    )
```

En nuestra implementación, utilizamos dos callbacks: CheckPointCallback y TensorBoardCallback.

CheckpointCallBack se encarga de almacenar los pesos de la red neuronal cada un número arbitrario de pasos, en este caso, los pesos se almacenan en la memoria cada 50.000 pasos en el directorio *models*. Esto es muy útil si queremos continuar un entrenamiento anterior, o compartir los pesos del modelo.

```
checkpoint_callback = CheckpointCallback(
    save_freq=50000,
    save_path="./models/",
    name_prefix="rl_model"
)
```

TensorBoardCallback, por otro lado, se encarga de almacenar en la memoria local diferentes métricas relacionadas con el entrenamiento en un formato específico que luego podremos ver en vivo en el navegador. Podemos ver un ejemplo de la interfaz de tensorboard en la figura 1.7

Tensorboard

Las métricas que podemos visualizar dependen del algoritmo de entrenamiento, pero para PPO podemos observar las siguientes:

Train KL: Mide la distancia entre políticas utilizando la divergencia KL. Cuanto mayor sea el valor de divergencia de KL, mayor será la distancia entre distribuciones de probabilidades entre la política actual y la anterior. El objetivo de PPO, a diferencia de otros algoritmos como DDPG, es minimizar y controlar la divergencia de KL entre dos políticas consecutivas. Esto asegura un entrenamiento estable, y que no haya una perdida de performance.

Train Clip Fraction: La fracción de veces que se utiliza el hiperparámetro de clip. Como se vio en la descripción de PPO, el algoritmo limita la magnitud del cambio entre dos politicas consecutivas para asegurar un entrenamiento estable. Se puede utilizar este gráfico como orientación sobre cómo elegir el valor del clip.

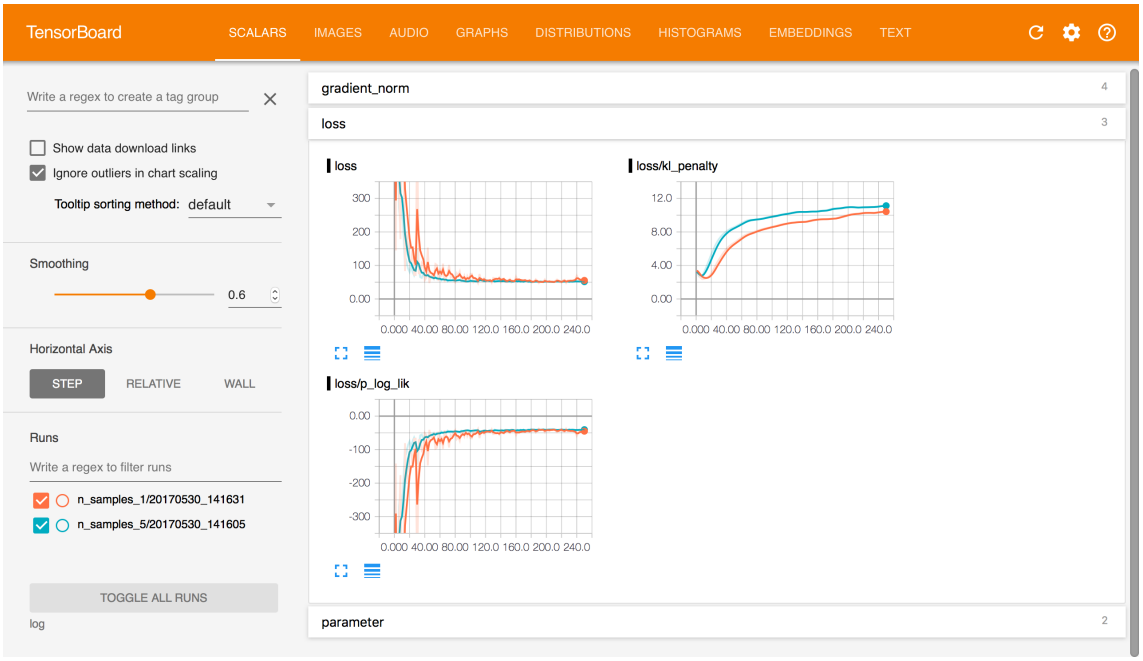

Fig. 1.7: Visualización de métricas con TensorBoard

Cumulative Reward: Promedio de la recompensa acumulada por episodio. La tendencia general de las recompensas debería aumentar con el tiempo en un entrenamiento exitoso. Se pueden esperar pequeños altibajos, y dependiendo de la complejidad de la tarea es posible que no se produzca un aumento significativo en la recompensa hasta que se hayan completado un gran número (cientos de miles o millones) de pasos en el proceso de entrenamiento.

Entropy: La entropía de una distribución de probabilidad mide cuán aleatoria es. En nuestro caso, mide la aleatoriedad de las decisiones que se toman. Al principio deberíamos permitir la exploración de nuevas acciones aleatoriamente, pero debe reducirse lentamente durante un entrenamiento exitoso.

Episode Length: La duración promedio de cada episodio. Deberia ir incrementando con el tiempo durante el entrenamiento.

Learning Rate: El tamaño del pasos que toma el algoritmo de descenso de gradiente durante el entrenamiento. Se pueden elegir distintas políticas de actualización de este parámetro. En nuestra implementación lo mantendremos constante durante todo el entrenamiento.

Policy Loss: El valor promedio de la función de pérdida de política. Se relaciona con el grado de cambio en la política. Durante un entrenamiento exitoso, el valor debería disminuir. Estos valores oscilan durante el entrenamiento.

Value Estimate: El promedio de la función de valor para cada estado visitado por el agente. Debería aumentar durante un entrenamiento exitoso. Estos valores deberían aumentar a medida que aumenta la recompensa acumulada. Corresponden a la cantidad de recompensa futura que el agente predice que recibirá en un momento dado para cada uno de los estados.

Value Loss: El valor promedio de la función de pérdida asociada a la función de valor. Se relaciona con la capacidad del modelo para predecir el valor en cada estado. Este valor

debería disminuir durante el entrenamiento.

1.2.4. Robot Operating System (ROS)

Introducción

ROS es un metasistema operativo de código abierto para robots. Proporciona los servicios que esperamos de un sistema operativo, como la abstracción de hardware, control de dispositivos de bajo nivel, implementación de funciones utilizadas frecuentemente, mensajería entre procesos y administración de paquetes. También proporciona herramientas y bibliotecas para obtener, crear, escribir y ejecutar código en varias computadoras. ROS es similar en algunos aspectos a frameworks de robótica como Player, YARP, Orocos, CARMEN, Orca, MOOS y Microsoft Robotics Studio.

El grafo de ejecución de ROS es una red de procesos (potencialmente distribuidos entre máquinas) que están débilmente acoplados mediante la infraestructura de comunicación de ROS. ROS implementa muchos estilos diferentes de comunicación entre procesos, incluida la comunicación síncrona estilo RPC a través de servicios y la transmisión de datos asíncrona a través de tópicos.

Grafo de ejecución de ROS

Fig. 1.8: Comunicacion entre dos nodos en ROS via tópicos y servicios.

Este grafo es una red peer-to-peer de procesos ROS que trabajan juntos para procesar datos. Los conceptos más importantes de gráficos computacionales de ROS son nodos, mensajes, servicios, y tópicos, que proporcionan datos al grafo de diferentes maneras.

Nodo: El concepto de nodos es central en la arquitectura de ROS. ROS está diseñado para ser modular; los sistemas de control de robots normalmente contienen varios nodos. Por ejemplo, un nodo para el ambiente del hexápodo que implementa la interfaz Gym vista anteriormente, otro nodo para realizar los cómputos de la cinemática inversa, otro nodo para proporcionar una vista gráfica del sistema, etc. Los nodos ROS se escriben utilizando bibliotecas cliente ROS como roscpp o rospy, dependiendo si se utiliza el lenguaje C++ o Python.

Mensajes: los nodos se comunican pasando mensajes. Un mensaje es simplemente una estructura de datos que contiene variables de distintos tipos. Admite tipos básicos estándar (enteros, punto flotante, booleano, etc.), así como tipos compuestos (muy parecidas a las estructuras C).

Tópicos: Los mensajes se envían a través de un sistema que sigue el patrón de publicador y suscriptor. Los nodos envían mensajes publicándolos en un tópico determinado. El tópico es el nombre utilizado para identificar el contenido del mensaje. Los nodos interesados en consumir ciertos datos se suscribirán al tópico apropiado. Un solo tópico puede tener múltiples publicadores y suscriptores simultáneos, y un solo nodo puede publicar y/o suscribirse a múltiples tópicos. En términos generales, los editores y los suscriptores desconocen la existencia del otro. Como mencionamos anteriormente, la idea central en ROS es desacoplar los nodos entre sí. Los tópicos estarán muy presentes en este trabajo con el objetivo de publicar las acciones que debe tomar nuestro agente al nodo de cinemática inversa, consumir datos acerca del estado del robot (la observación) en un determinado momento, etc.

Servicios: el modelo de publicador y suscriptor es un paradigma de comunicación muy flexible, pero su transmisión unidireccional de muchos a muchos no es siempre la adecuada, en especial cuando estamos interesados en realizar llamadas sincrónicas. Para esto, ROS introduce el concepto de un servicio, que está definido por un par de estructuras de mensajes: una para la solicitud y otra para la respuesta. Un nodo proveedor proporciona un servicio por nombre y los clientes consumen el servicio enviando un mensaje de solicitud y esperando una respuesta. Las bibliotecas cliente de ROS normalmente presentan esta interacción al programador como si fuera una llamada a un procedimiento remoto. En este trabajo utilizaremos servicios para temporalmente pausar y despausar la dinámica del sistema.

Nodo Mastro: ROS cuenta con un nodo Maestro que proporciona un registro de nombres y búsqueda de procesos dentro del gráfico de cómputo. Sin un maestro, los nodos no podrían encontrarse entre sí, intercambiar mensajes ni invocar servicios. El nodo maestro almacena información de registro de tópicos y servicios. Los nodos se comunican con el Maestro para reportar su información de registro. Cuando estos nodos se comunican con el nodo maestro, pueden recibir información sobre otros nodos registrados y establecer conexiones. Los nodos se conectan directamente a otros nodos; el maestro solo proporciona información de búsqueda, de forma muy parecida a un servidor DNS. Un nodo que se suscriba a un tópico solicitará una conexión al nodo que publica el tópico y la conexión se establecerá mediante el protocolo de conexión acordado. El protocolo más utilizado en ROS se llama TCPROS, que utiliza sockets TCP/IP estándar.

1.2.5. Gazebo

Gazebo es un simulador de robots 3D de código abierto utilizado en conjunto con ROS. Gazebo simula la física del mundo real en simulaciones realistas. La simulación de robots puede ahorrar mucho tiempo y dinero porque nos permite entrenar y probar el róbot hexápodo sin requerir de la experimentación con el robot real, que implica ponerlo operativo, cargar las baterías (que tienen una vida útil) y potenciales daños durante el proceso de entrenamiento. Además, Gazebo nos permite crear simulaciones utilizando múltiples motores de física de alto rendimiento como ODE, Simbody, Bullet y DART. Nos ayuda a replicar la gravedad, la fricción, el torque y cualquier otra condición del mundo real que pueda afectar el entrenamiento.

2. TRABAJOS RELACIONADOS

Existen algunos trabajos en el estado del arte que han explorado el uso de RL específicamente en robots hexápodos. Trabajos como [7] usan RL en simulación para que un hexápodo aprenda a caminar, pero sólo en terreno plano. En [8] se implementa RL para ajustar los parámetros de un generador de patrón central (CPG por sus siglas en inglés) en un robot real, pero nuevamente los terrenos utilizados son esencialmente planos. En [9] el robot es solo simulado y el terreno tiene irregularidades, pero no muy pronunciadas ni diversas. En [10] se consideran terrenos más desafiantes y se propone un sistema de dos etapas: una que aprende por DRL (Deep Reinforcement Learning) una caminata óptima para cada terreno dentro de un número de posibles tipos y un clasificador que aprende a predecir el tipo de terreno para seleccionar la política experta correspondiente. El trabajo presenta resultados muy interesantes, pero únicamente en simulación y utilizando un modelo de robot simplificado a figuras geométricas sencillas, lo que impide asegurar que el resultado sea trasladable a un robot hexápodo real.

En [11] se estudia la adaptabilidad de los robots hexápodos en terrenos irregulares con un enfoque en mejorar la estabilidad y la postura del robot. En dicho trabajo, se propuso un mecanismo de control de postura que ajusta la orientación y altura del cuerpo del hexápodo en función de la topología del terreno, basado en estimaciones de la pendiente del suelo, las posiciones de las extremidades y las especificaciones físicas del robot. Este método fue evaluado utilizando simulaciones computacionales en Gazebo, con buenos resultados para mantener la estabilidad tanto en rampas como en superficies irregulares, similares a las de nuestro trabajo. Sin embargo, este trabajo asume conocimiento detallado de los parámetros físicos del hexápodo y no explora el potencial de enfoques basados en aprendizaje automático para mejorar la adaptabilidad en tiempo real. En contraste, nuestro trabajo emplea técnicas de aprendizaje por refuerzo que no dependen de modelos cinemáticos predefinidos y podrían adaptarse a distintos tipos de hardware.

En [12], se abordó el problema del control de un robot hexápodo aplicando aprendizaje por refuerzo profundo mediante una estructura modular y descentralizada. Este enfoque está inspirado en el control locomotor de los insectos, donde cada pata tiene su propio controlador individual. Los resultados iniciales mostraron que este enfoque descentralizado mejora la capacidad de aprendizaje de tareas como ponerse de pie y caminar, aunque el robot aún presenta dificultades para mantener el torso elevado durante la marcha. En contraste, nuestro trabajo utiliza un enfoque centralizado, en el que una única red neuronal controla el robot en su totalidad. Esta decisión fue motivada por las limitaciones del hardware disponible y el tiempo de entrenamiento requerido para entrenar seis módulos individuales. Este enfoque simplifica la arquitectura y es más adecuado cuando se enfrenta a restricciones computacionales.

Por otro lado, existen más trabajos realizados para robots cuadrúpedos, donde hay múltiples ejemplos de sistemas llevados al mundo real, con resultados sumamente interesantes. En [13] se entrena con RL una política que luego se transfiere a un robot cuadrúpedo real y le permite moverse en escaleras, terrenos con agujeros y rocas. Aún más impresionante son los resultados obtenidos en [14], donde logran que un cuadrúpedo ANYmal pueda atravesar escaleras, terreno boscoso inclinado, vegetación, nieve entre otros terrenos. No obstante, ambos trabajos hacen uso de un sistema de visión, que escapa al alcance de

este trabajo, donde queremos obtener un sistema de locomoción sin hacer uso de sensores exteroceptivos. Por otro lado, en [15] obtienen resultados similares para un robot ANYmal utilizando únicamente medidas propiocépticas, entrenando en simulación y pudiendo transferir la política aprendida al robot real sin ajuste adicional. Para ello proponen una estrategia maestro-estudiante, que consiste en entrenar en simulación un robot maestro con acceso a información privilegiada obtenida del simulador y luego usar la pólitica obtenida para entrenar por aprendizaje supervisado un robot estudiante que solo tiene acceso a las mediciones que estarían disponibles en el robot real.

En [16], se propuso entrenar robots cuadrúpedos directamente en el mundo real, sin simuladores. Este enfoque le permite a los robots aprender de manera online a partir de una cantidad limitada de interacciones con el entorno. Los experimentos incluyeron tareas como levantarse, caminar y manipular objetos. A pesar de los avances, este enfoque utiliza una función de recompensa muy simplificada que únicamente maximiza la velocidad lineal, ignorando otros aspectos cruciales como la suavidad del movimiento, la velocidad angular y la estabilidad a largo plazo. Además, el entrenamiento únicamente en el mundo físico requiere una infraestructura de hardware robusta y limita el número de iteraciones que pueden realizarse en un período de tiempo. Por otro lado, el entrenamiento prolongado en hardware puede generar desgaste en los robots, lo que podría requerir intervención humana o reparaciones, incrementando los costos totales de los experimentos.

En este trabajo presentamos un desarrollo que sirve como módulo maestro de un sistema de locomoción para robots hexápodos utilizando únicamente mediciones propioceptivas y Aprendizaje por Refuerzo Profundo, basado en el enfoque maestro-alumno que se presenta en [15]. Específicamente, el sistema está diseñado para el robot PhantomX Mark II. A diferencia de los trabajos antes mencionados, hacemos uso de herramientas ampliamente utilizadas por la comunidad: Gazebo [17] como plataforma de simulación, ROS [18] para comunicación entre los distintos módulos del sistema y la librería Stable Baselines 3 [19] para los algoritmos de Aprendizaje por Refuerzo Profundo. Los terrenos utilizados para el entrenamiento fueron terrenos de colinas, generados con ruido de Perlin, seleccionados adaptativamente con un filtro de partículas para aumentar progresivamente la dificultad del terreno con el que se entrena.

3.1. Definición del Problema

Como mencionamos anteriormente, nuestro objetivo es enseñar a un robot hexápodo a caminar en terrenos irregulares utilizando aprendizaje por refuerzo. Vamos a definir formalmente nuestro problema como un Proceso de Decisión de Markov dado por una cinco-upla $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, >$ tal que:

- $\mathcal{S}$ es un conjunto finito de estados
- $\mathcal{A}$ es un conjunto finito de acciones.
- $\mathcal{P}$ representa la dinámica del sistema.
- $\mathcal{R}$ es la función de recompensa.
- γ tal que $0 \leq \gamma \leq 1$ es el factor de descuento.

En este problema $\mathcal{P}$ es claramente desconocido, y el valor de γ sera detallado en la sección de hiperparametros del algoritmo.

3.1.1. Conjunto de Estados

El vector de observaciones $\mathcal{S}$ es la entrada a la red neuronal, que se construye a partir del comando de velocidad, mediciones del simulador y variables internas del sistema. Se actualiza a $50Hz$, que es la frecuencia de trabajo del sistema de control de alto nivel. Para el robot maestro implementado en este trabajo el vector de estado está compuesto por:

- Dirección de velocidad lineal deseada $({}^{B}_{IB}\hat{v}_d)_{xy}$).
- Dirección de velocidad angular deseada $({}^{B}_{IB}\hat{\omega}_d)_z$).
- Vector de gravedad g.
- Velocidad angular de la base ${}^{B}_{IB}\omega$).
- Velocidad lineal de la base ${}^{B}_{IB}v$).
- Posición/velocidad de cada articulación $(\theta_i, \dot{\theta}_i)$.
- Fases para la generación de trayectorias $(sin(\phi), cos(\phi))$.
- Frecuencias para la generación de trayectorias $(\dot{\phi})$.
- Frecuencia base (f_0).
- Últimos dos valores de error de posición de cada articulación $e_{\theta_{t-1}}, e_{\theta_{t-2}}$.
- Últimos dos valores de velocidad de cada articulación $\dot{\theta}_{t-1}, \dot{\theta}_{t-2}$.

- Últimos dos valores de posición deseada de cada pata ($(r_{i,d})_{t-1,t-2}$).
- Normal del terreno en cada pata n_i.
- Altura de cada pata y de 8 puntos alrededor sobre un círculo de radio $10cm$.
- Fuerzas de contacto de cada pata: F_{c_i}
- Estados de contacto de cada pata: c_i
- Estados de contacto de cada fémur: c_{femur_i}
- Estados de contacto de cada tibia: c_{tibia_i}
- Coeficientes de fricción pata-suelo: μ_i

3.1.2. Conjunto de Acciones

El conjunto de acciones $\mathcal{A}$ esta dado por la frecuencia de la trayectoria f_i y un diferencial de movimiento Δr_i para cada pata, como es detallado en la sección 3.2.2. Estos valores se utilizan junto con el modulo de generación de trayectorias para calcular las posiciones objetivos de las patas, que luego el modulo de cinemática inversa transforma en los ángulos objetivos de cada articulación.

Para dar un ejemplo concreto, la red neuronal nos podría llegar a devolver que las frecuencias para cada una de las seis patas serán: $f_1 = 0{,}3$, $f_2 = 0{,}8$, $f_3 = 0{,}2$, $f_4 = 0{,}5$, $f_5 = 0{,}1$ y $f_6 = 0{,}3$. Adicionalmente, nos retorna el diferencial de movimiento para cada una de las patas $\Delta r_1 = 0{,}5$ $\Delta r_2 = 0{,}3$ $\Delta r_3 = 0{,}1$ $\Delta r_4 = 0{,}8$ $\Delta r_5 = 0{,}9$ $\Delta r_6 = 1$.

La red neuronal siempre nos devuelve resultados en el rango $[0, 1]$, y luego los podemos escalar por una constante. En nuestro caso, escalamos los Δ_r utilizando la constante 5 para evitar movimientos muy bruscos en las patas.

3.1.3. Función de Recompensa

La función de recompensa $\mathcal{R}$ utilizada en este trabajo es:

$$R = 0{,}05r_{vl} + 0{,}05r_{va} + 0{,}05r_b + 0{,}01r_{ep} + 0{,}02r_c + 0{,}025r_s \tag{3.1}$$

donde los términos se explican a continuación, considerando que $({}^{B}_{IB}v)$ es la velocidad lineal de la base respecto del sistema global expresada en coordenadas de la base, $({}^{B}_{IB}\hat{v}_d)$ es el correspondiente comando de velocidad lineal normalizado, análogo para la velocidad angular ω, y el subíndice xy indica la proyección ortogonal sobre este plano mientras que z indica la proyección ortogonal sobre esta dirección.

- Recompensa por velocidad lineal (r_{vl}):

$$r_{vl} = \begin{cases} e^{-\alpha_{vl}(v_p - 0{,}1)^2} & v_p < v_{threshold} \\ 1 & v_p \geq v_{threshold} \end{cases}$$

  donde $v_p = ({}^{B}_{IB}v)_{xy}({}^{B}_{IB}\hat{v}_d)_{xy}$ es la proyección ortogonal de la velocidad lineal de la base en xy sobre la dirección del comando de velocidad lineal. El objetivo de este término es que el robot camine en la dirección indicada por el comando de velocidad lineal, maximizando dicha proyección.

- Recompensa por velocidad angular (r_{va}):

$$r_{va} = \begin{cases} e^{-\alpha_{va}(\omega_p - 0{,}1)^2} & \omega_p < \omega_{threshold} \\ 1 & \omega_p \geq \omega_{threshold} \end{cases}$$

  donde $\omega_p = ({}^B_{IB}\omega)_z({}^B_{IB}\hat{\omega}_d)_z$ es la proyección ortogonal de la velocidad angular de la base en z sobre la dirección del comando de velocidad angular. El objetivo de este término es que el robot rote en la dirección indicada por el comando de velocidad angular maximizando dicha proyección. En los experimentos que realizamos, mantuvimos la velocidad angular nula para evitar rotaciones.

- Recompensa por movimiento de la base (r_b):

$$r_b = e^{-\alpha_b v_0^2} + e^{-\alpha_b ||({}^B_{IB}\omega)_{xy}||^2}$$

  donde $v_0 = ||({}^B_{IB}v)_{xy} - v_p({}^B_{IB}\hat{v}_d)_{xy}||$. El objetivo de este término es penalizar el movimiento ortogonal respecto al vector comando.

- Recompensa por elevación de patas (r_{ep}):

$$r_{ep} = \sum_{i \in I_{swing}} \chi_{\mathcal{F}_{clear}} / |I_{swing}|$$

  donde I_{swing} es el conjunto de patas en movimiento (en inglés swing), χ es la función característica y $\mathcal{F}_{clear}$ es el conjunto de patas cuya posición es mayor o igual en altura que los 8 puntos medidos a su alrededor. El objetivo de este término es que las patas puedan anticiparse a una elevación en la altura del terreno.

- Recompensa por colisión (r_c):

$$r_c = -|I_{c,cuerpo} \setminus I_{c,patas}|$$

  es decir, sumamos las partes del robot que tienen una fuerza ejercida sobre ellas. La magnitud de las derivadas de segundo orden de las posiciones objetivo se penalizan para que las trayectorias generadas sean más suaves.

- Recompensa por suavidad (r_s):

$$r_s = -\sum_i ||(r_{i,d})_t - 2(r_{i,d})_{t-1} + (r_{i,d})_{t-2}||$$

  donde $(r_{i,d})_t$ es la posición deseada para la pata i-ésima en el instante t. El objetivo de este término es suavizar el movimiento del robot.

Los parámetros de la función de recompensa, incluyendo las constantes que acompañan cada término de la ecuación (3.1), se ajustaron manualmente para obtener mejores resultados. En función de los ensayos realizados, se definió usar $\alpha_{vl} = \alpha_{va} = 100$, $\alpha_b = 500$ y $v_{threshold} = \omega_{threshold} = 0{,}2$.

3.2. Control

3.2.1. Esquema del robot PhantomX Mark II

Para este trabajo contamos con un robot hexápodo PhantomX Mark II. Los modelos 3D del robot fueron obtenidos del repositorio de Github citado en [20].

El robot se describe mediante un archivo URDF. El modelo URDF, que significa Unified Robot Description Format (Formato Unificado de Descripción de Robot), es un estándar utilizado en robótica para describir la estructura y cinemática de un robot en un formato XML (Lenguaje de Marcado Extensible). Esta descripción proporciona información detallada sobre la geometría, cinemática, dinámica y otros aspectos esenciales del robot, lo que facilita su simulación, visualización y control.

Cada pata de este robot hexápodo se compone de tres partes distintas, identificadas como coxa, fémur y tibia. Estas se encuentran conectadas entre sí por medio de tres articulaciones representadas por las variables angulares $(\theta_1, \theta_2, \theta_3)$, como se puede observar en la figura 3.9. Este diseño modular y articulado permite una movilidad precisa y coordinada del robot, lo cual resulta esencial para la ejecución eficiente de los patrones de marcha o caminata.

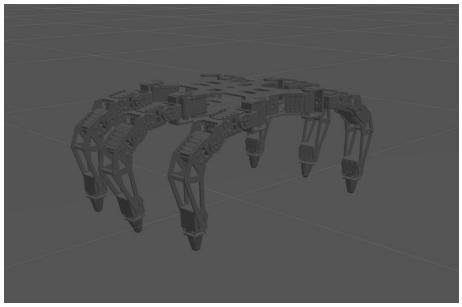

Fig. 3.1: Robot hexápodo PhantomX

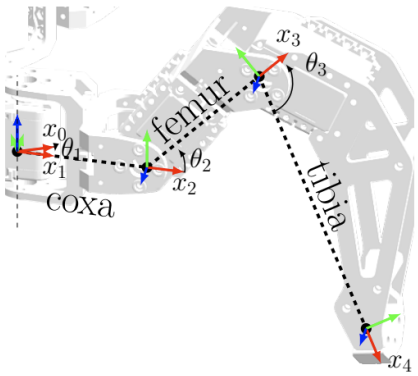

Fig. 3.2: Esquema de una pata

3.2.2. Estructura General de Control

En este trabajo nuestro objetivo es enseñar a un robot hexápodo a caminar en terrenos irregulares usando únicamente sensores propioceptivos. El robot utilizado para este trabajo es el PhantomX Mark II, cuyo modelo URDF y esquema fue descripto en la sección 3.2.1

La red neuronal que se entrena mediante aprendizaje por refuerzo profundo modifica la trayectoria de cada pata fijada *a priori* por el controlador de bajo nivel, que por sí solo mantiene al robot caminando en el lugar en un terreno plano, para que se adapte al terreno irregular y siga el comando de velocidad deseado.

Para ello se modifican la frecuencia de la trayectoria de cada pata, f_i, y la posición final de referencia adicionando un diferencial, Δr_i, como se explica a continuación.

La estructura general del control, mostrada en la Figura 3.3, se compone, en primer lugar, de una red neuronal, que recibe comandos de velocidad lineal y angular, y cuya

salida son los valores de frecuencia f_i de cada pata y los diferenciales de movimiento Δr_i. Los valores de frecuencia sirven de entrada al bloque de generación de trayectoria que define el movimiento a realizar por cada pata. El valor de posición objetivo para cada pata es el obtenido por el generador de trayectoria más el diferencial de movimiento Δr_i determinado por la red neuronal. Este valor final se procesa con las ecuaciones de cinemática inversa para obtener las posiciones deseadas para cada articulación, que son enviadas a los controladores PID de cada articulación del robot. Finalmente, el robot actúa y el simulador devuelve las nuevas observaciones.

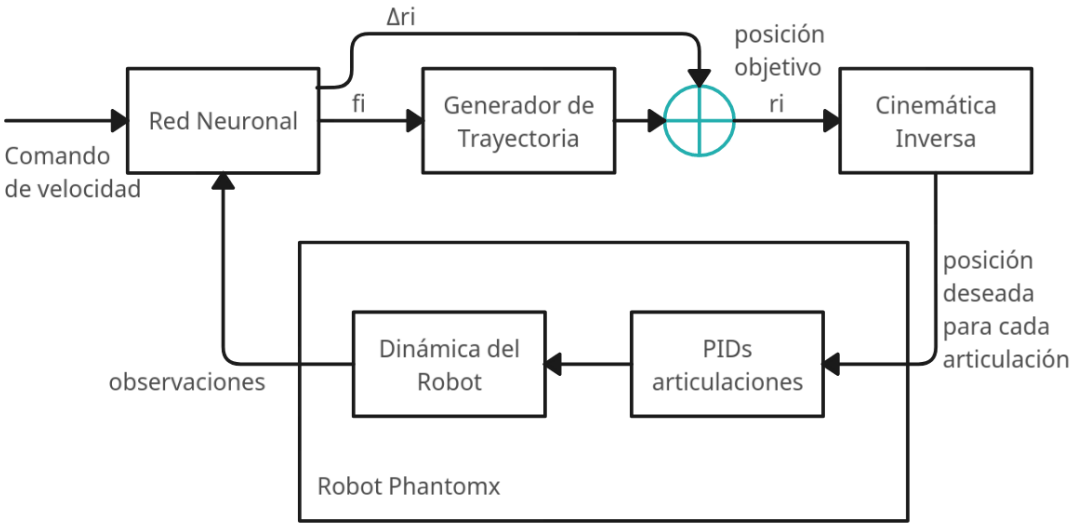

Fig. 3.3: Diagrama en bloques del sistema de control del robot.

3.2.3. Modelo Cinemático

Una vez que el generador de trayectorias nos devuelve cuales son las posiciones objetivos de cada una de las patas, debemos calcular los ángulos correspondientes para cada una de las articulaciones del robot. La responsabilidad de calcular dichos ángulos es del sistema de cinemática inversa.

Matemáticamente, es posible establecer la posición de la pata a partir de los ángulos de coxa, femur y tibia (θ_C, θ_F y θ_T, respectivamente) conociendo las longitudes de cada parte (LC, LF y LT, respectivamente) haciendo uso del eje auxiliar r_{xy}:

$$r_{xy} = LC + LFcos(\theta_F) + LTcos(\theta_F + \theta_T)$$

$$z = LFsen(\theta_F) + LTsen(\theta_F + \theta_T)$$

$$x = r_{xy}cos(\theta_C)$$

$$y = r_{xy}sen(\theta_C)$$

A partir de estas relaciones podemos obtener las ecuaciones de cinemática inversa:

$$\theta_C = atan2(y, x)$$

$$u = \sqrt{x^2 + y^2} - LC$$

$$c2 = cos(\theta_T) = \frac{u^2 + z^2 - LF^2 - LT^2}{2LF\ LT}$$

$$s2 = sen(\theta_T) = -\sqrt{1 - c2^2}$$

$$c1 = cos(\theta_F) = \frac{u(LF + LT\ c2) + z\ LT\ s2}{u^2 + z^2}$$

$$s1 = sen(\theta_F) = \frac{z(LF + LT\ c2) - u\ LT\ s2}{u^2 + z^2}$$

$$\theta_F = atan2(s1, c1)$$

$$\theta_T = atan2(s2, c2)$$

Como estas ecuaciones asumen cierta orientación del sistema de coordenadas de cada articulación y que todos los ángulos en cero corresponden a la pata estirada recta, los valores de comando real a enviar al simulador son: $\theta_{coxa} = \theta_C$, $\theta_{femur} = -\theta_F$ y $\theta_{tibia} = \theta_T + \pi/2$.

3.2.4. Control de Bajo Nivel

El control de bajo nivel consta de varias etapas. La primera es la generación de trayectoria de referencia para cada pata. En cada instante de tiempo la fase de la pata i está dada por:

$$\phi_i = (\phi_{i,0} + (f_0 + f_i)t)(mod\ 2\pi)$$

donde $\phi_{i,0}$ es la fase inicial, f_0 es una frecuencia base común a todas las patas y f_i es la frecuencia de offset de la i-ésima pata. La fase indica etapa de contacto para $\phi_i \in [0, \pi)$ y movimiento para $\phi_i \in [\pi, 2\pi)$. Dada la fase ϕ_i, la posición objetivo de la pata será $r_i = F(\phi_i) + \Delta r_i$, donde Δr_i es la salida de la red neuronal, así como f_i, y la función F viene dada por:

$$F(\phi_i) = \begin{cases} (h(-2k^3 + 3k^2) - 0{,}5)^{H_i Z} & k \in [0, 1] \\ (h(2k^3 - 9k^2 + 12k - 4) - 0{,}5)^{H_i Z} & k \in [1, 2] \\ -0{,}5^{H_i Z} & sino \end{cases}$$

donde $k = 2(\phi_i - \pi)/\pi$, h es un parámetro para la altura máxima del paso y H_i es un sistema de coordenadas de referencia colocado a una distancia del marco de coxa dada por la postura de parada de referencia, cuya orientación es siempre horizontal, con el eje x paralelo a la proyección ortogonal del eje x de la base del robot sobre el plano xy, como se puede ver en la Figura 3.4. De esta forma si $f_i = 0$ y $\Delta r_i = \overrightarrow{0}$ para todo i, el robot se mantiene caminando en el lugar.

Una vez obtenido los r_i en sus sistemas de coordenadas H_i, se convierte al sistema de coordenadas de coxa de cada pata y se usan las ecuaciones de cinemática inversa para obtener los valores de ángulos a usar como comandos de los controles de posición de cada articulación (los controladores PID ya están incluidos en el modelo URDF del robot PhantomX).

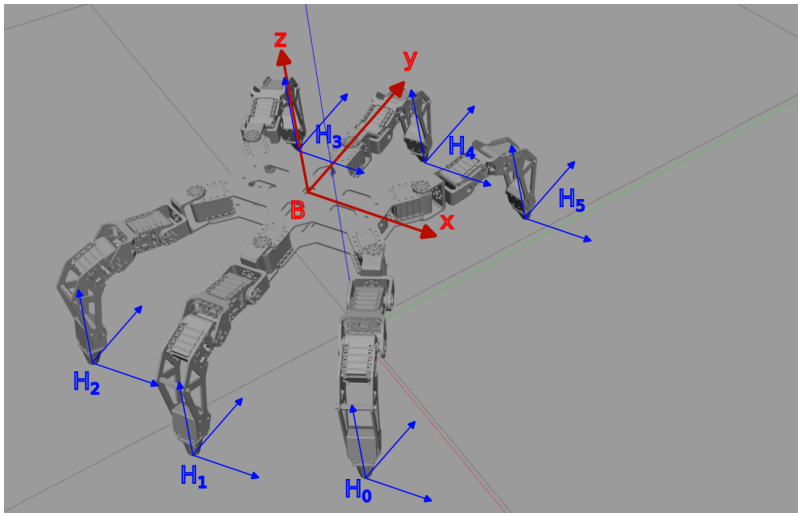

Fig. 3.4: Robot PhantomX con los sistemas de coordenadas de referencia H_i (azul) y el sistema de coordenadas de la base B (rojo).

3.3. Generación de Terrenos

Para que el robot pueda adaptarse y transitar distintos tipos de entorno, es necesario presentarle una dificultad creciente durante el proceso de entrenamiento, tales como irregulares o ciertos obstáculos en el terreno. Esto hará que el robot se exponga a distintos tipos de situaciones que pueden aparecer en el mundo real y aprenda cómo reaccionar ante ellas, resultando en un controlador más robusto.

Para generar terrenos aleatorios de manera eficiente, utilizamos la técnica de generación procedural de terrenos. Estas técnicas se basan en la creación de terrenos mediante algoritmos en lugar de generarlos de forma manual, y son muy utilizadas en la computación gráfica, o incluso en la generación de terrenos para videojuegos. Para poder crear una gran cantidad de terrenos de manera eficiente, utilizamos un conjunto de parámetros $c_T \in C$ que definirán la estructura y geometría del terreno. En este trabajo, utilizamos terrenos montañosos que podemos ver en las figuras 3.5, 3.6 y 3.7.

3.3.1. Ruido de Perlín

El Ruido de Perlín es un algoritmo muy conocido que se utiliza con frecuencia en la generación de contenido procedural. Es especialmente útil para videojuegos y generación de gráficos por computadora. Fue creado por Ken Perlin, quien fue otorgado un premio en reconocimiento de su gran invento. Este algoritmo se puede utilizar tanto en una dimensión, como en múltiples dimensiones. En este trabajo, lo utilizaremos para generar un mapa de alturas en dos dimensiones, que luego será interpretado por el motor gráfico de Gazebo para generar terrenos en tres dimensiones. Un mapa de alturas es una matríz M en dos dimensiones, tal que $M[x][y] \in [0, 1]$. Cada valor del mapa se interpreta como una altura en el terreno.

El algoritmo de Perlín, a diferencia de un generador de números aleatorios, genera una secuencia de valores pseudoaleatorios en el rango $[0, 1]$. Un algoritmo de generación de números aleatorios, genera valores que no están relacionados entre sí de ninguna manera:

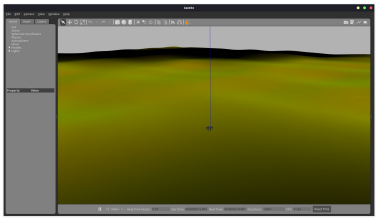

Fig. 3.5: a=0.1 r=0.1 f=0.1

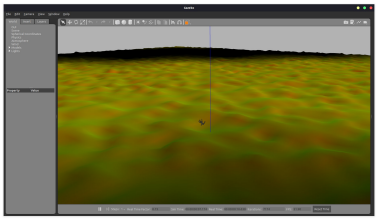

Fig. 3.6: a=0.3 r=0.5 f=0.5

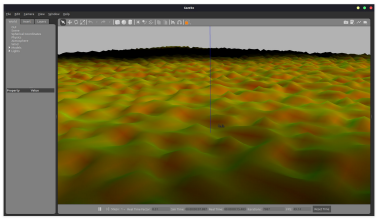

Fig. 3.7: a=0.5 r=1 f=0.5

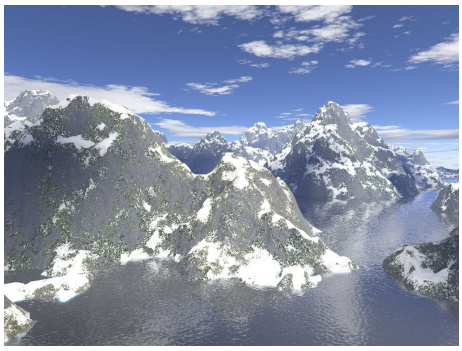

Fig. 3.8: Un terreno generado utilizando Ruido de Perlín

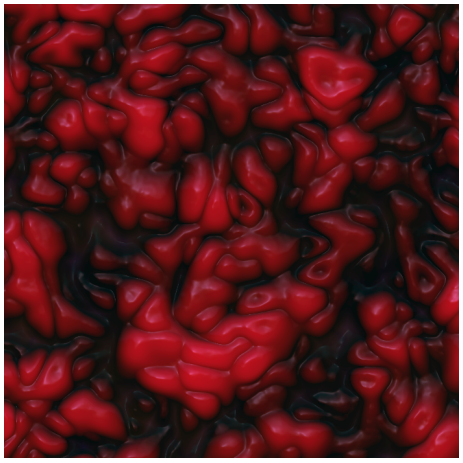

Fig. 3.9: Una superficie orgánica generada utilizando Ruido de Perlín

cada valor es independiente del anterior. Esto se puede observar en la figura 3.10, en donde generamos una señal aleatoria.

En cambio, el ruido de Perlin tiene una apariencia más natural, ya que genera una secuencia de números pseudoaleatorios que están relacionados entre sí. En la figura 3.11 podemos ver ruido de Perlín que fue generado a lo largo del tiempo, con el eje x representando la variable temporal. Lo que lo distingue a un generador de números aleatorios, es que dados dos tiempos t_1 y t_2 cercanos entre sí, $perlin(t_1)$ y $perlin(t_2)$ tomarán valores similares.

En nuestro trabajo generaremos una matriz de dimensión 129×129, en donde para cada coordenada (x, y) se genera un valor pseudoaleatorio utilizando la función de Perlin. A continuación, se muestra una versión inicial del algoritmo para generar terrenos montañosos:

```
def octaves_perlin(width=129, height=129):
```

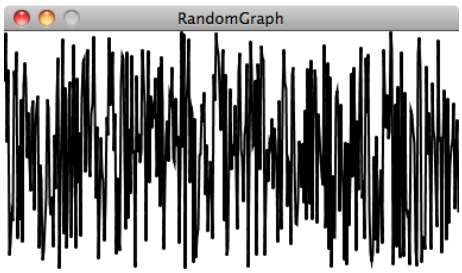

Fig. 3.10: Señal en una dimensión generada aleatoriamente

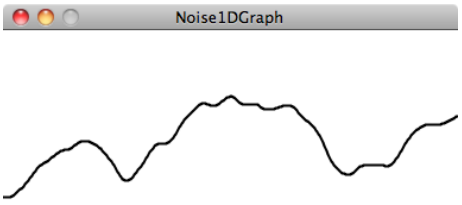

Fig. 3.11: Señal en una dimensión, generada utilizando Ruido de Perlín

```
noise = PerlinNoise(octaves=1)
heightMap = np.zeros((height, width))
for y in range(height):
    for x in range(width):
        heightMap[y][x] = noise([x,y])
return heightmap
```

El resultado de este algoritmo es una matriz con valores en el rango $[0, 1]$ como se puede observar en la figura 3.12

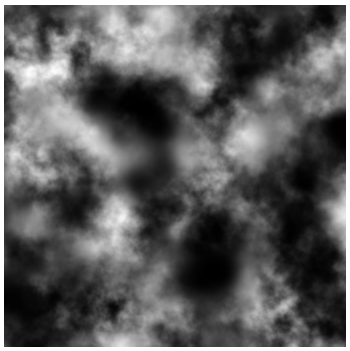

Fig. 3.12: Mapa de alturas generado utilizando Ruido de Perlín

Cada uno de los valores de la matriz en la posición (x, y) estará dentro del rango $[0, 1]$ y se interpretará por el motor de Gazebo como la altura del terreno en la coordenada (x, y).

Para que los terrenos generados tengan distintas dificultades, utilizamos tres parámetros para caracterizar los terrenos generados mediante el Ruido de Perlín: rugosidad r, frecuencia f, y amplitud a.

La rugosidad define el grado de irregularidades que posee una superficie. Para generar picos de distintas alturas en los terrenos, se genera un ruido uniforme $u \sim U(-r, r)$ que es sumado al ruido de Perlín.

La frecuencia hace referencia al periodo con la que se toman las muestras, y la amplitud define el rango en el cual pueden estar los valores. En la figura 3.13, se muestran distintas señales generadas utilizando Ruido de Perlín para distintas combinaciones de amplitud y frecuencia. Cada una de esas señales se conoce como un Octave.

Como se puede observar, mayor amplitud resulta en terrenos con diferencias de altu-

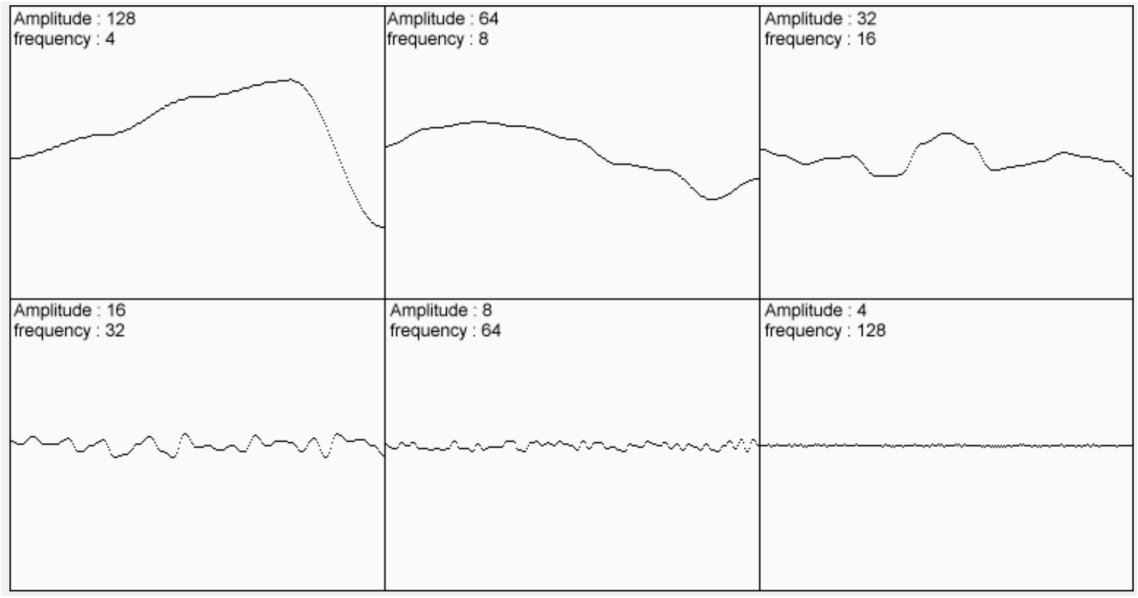

Fig. 3.13: Ejemplos de ruido con diferentes frecuencias y amplitudes

ras más significativas, mientras que a mayor frecuencia tendremos terrenos mas suaves. Podemos obtener la señal final sumando cada una de las señales individuales, obteniendo el resultado que se muestra en la figura 3.14. El algoritmo final para generar el ruido de perlin en la posición (x, y) del mapa de alturas es el siguiente:

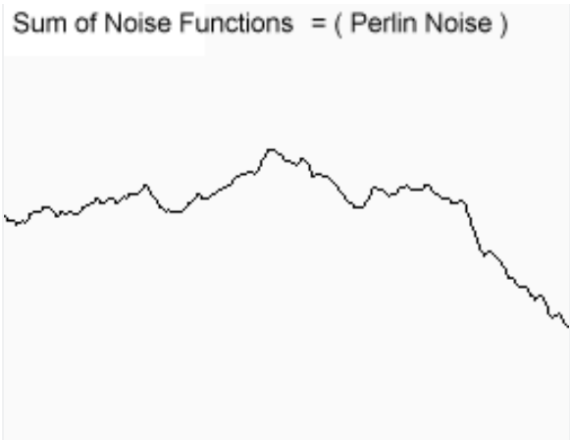

Fig. 3.14: La suma de las 7 señales de la figura 3.13

```
def octave_perlin_at_coordinate(noise, octaves, persistence, roughness,
frequency, amplitude, lacunarity, x, y):
    total = 0
    maxValue = 0
    # Loop through every octave.
    for i in range(octaves):
        total += amplitude (noise([x frequency, y frequency])
        + random.uniform(-roughness, roughness))
```

```
        # Update subsequence values
        maxValue  += amplitude
        amplitude *= persistence
        frequency *= lacunarity
    # Normalize value in range [0,1]
    return total/maxValue
```

3.4. Entrenamiento con Currícula de Terrenos

Como vimos en la sección anterior, podemos utilizar Ruido de Perlín para generar distintos tipos de terrenos. Algunas preguntas que surgen naturalmente son: ¿Cómo seleccionamos los parámetros utilizados para generar el terreno? ¿Cada cuantos episodios generamos un nuevo terreno? ¿Cómo determinamos si el robot está en condiciones de aprender sobre un terreno de mayor dificultad? En esta sección vamos a responder a estas preguntas y presentar dos metodologías que utilizamos durante el entrenamiento: Aleatoria y Adaptativa.

3.4.1. Generación Aleatoria de Terrenos

La metodología de generación aleatoria de terrenos va a generar un nuevo terreno utilizando parámetros aleatorios para la rugosidad r, frecuencia f, y amplitud a cada vez que un episodio haya finalizado. A continuación ilustramos este concepto con un fragmento de código:

```
def reset():

                        ...

    new_terrain_generated = octaves_perlin(
        roughness= np.random.uniform(low=0.0, high=0.05, size=1)
        frequency= np.random.uniform(low=0.2, high=1.0, size=1)
        amplitude= np.random.uniform(low=0.2, high=3.0, size=1)
    spawn_model_client(new_terrain_generated)
```

Esta metodología se caracteriza por ser muy sencilla de implementar, pero puede resultar en un entrenamiento lento. Esto se debe a que dicha metodología no tiene en cuenta ninguna métrica relacionada al aprendizaje del robot. Es decir, puede ocurrir que se genere un terreno demasiado difícil de atravesar para un robot que recién está comenzando el proceso de entrenamiento.

Una posible mejora a la metodología aleatoria podría ser computar una métrica para cuantificar la dificultad que presenta el robot para atravesar un terreno en particular. De esta forma, podríamos comenzar el entrenamiento con terrenos sencillos, y a medida en que el robot logra atravesarlo y la métrica de dificultad disminuye, se incrementa levemente la dificultad del terreno. Esto nos permitiría transicionar de forma suave entre los terrenos, permitiendo una mayor velocidad en el entrenamiento. A continuación, estudiaremos dicha mejora utilizando una metodología adaptativa.

3.4.2. Generación Adaptativa de Terrenos

A comparación con la generación aleatoria de terrenos, queremos modificar gradualmente los parámetros con los que se generan los terrenos de manera tal que nuestra política pueda mejorar continuamente las habilidades de locomoción y generalizar a nuevos entornos. Para esto, vamos a necesitar definir una métrica para cuantificar la dificultad del terreno para nuestro agente, llamada la transitabilidad.

Transitabilidad

La transitabilidad de un terreno generado mide la tasa de éxito al atravesarlo. La podemos utilizar para evaluar el progreso del aprendizaje de un agente. Notar que podríamos haber utilizado la función de recompensa, pero esta métrica resulta ser más intuitiva ya que consta de un único término acotado.

En primer lugar, definimos la función de transitabilidad en un paso v para la transición desde el estado s al estado s_{t+1}.

$$v\left(s_t, a_t, s_{t+1}\right) = \begin{cases} 1 & v_{pr}\left(s_{t+1}\right) > 0{,}2 \\ 0 & v_{pr}\left(s_{t+1}\right) < 0{,}2 \vee \text{ termination} \end{cases}$$

donde$v_{pr}(s_{t+1})$ es el producto interno entre el vector velocidad de la base y la dirección del comando en el tiempo $t+1$, que indica la velocidad con la cual el agente logra seguir la velocidad indicada por el comando.

Si la política π puede moverse en la dirección dada más rápido que $0,2$ m/s, vamos a considerar al terreno transitable en esta dirección. El umbral es un hiperparámetro; $0,2$ m/s es aproximadamente un tercio de la velocidad máxima de nuestro robot.

La transitabilidad de un terreno c_T bajo la política π se define como el siguiente valor esperado:

$$\mathrm{Tr}\left(c_T, \pi\right) = \mathbb{E}_{\tau\sim\pi}\left\{v\left(s_t, a_t, s_{t+1} \mid c_T\right)\right\} \in [0{,}0, 1{,}0]$$

donde ξ son trayectorias generadas por la política π.

El objetivo del método de generación adaptativa de terrenos es obtener una distribución para muestrar parámetros c_t con transitabilidad en un rango medio $\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]$. La razón es generar terrenos que no sean ni demasiado fáciles ni demasiado difíciles para el agente en el momento dado del entrenamiento.

3.4.3. Deseabilidad

Dada una política π y un terreno c_T definimos la deseabilidad de un terreno $\mathrm{Td}\left(c_T, \pi\right)$ como la probabilidad de que su transitabilidad esté en el rango deseado.

$$\begin{aligned} \mathrm{Td}\left(c_T, \pi\right) &:= \Pr\left(\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]\right) \\ &= \mathbb{E}_{\tau\sim\pi}\left\{\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]\right\} \end{aligned}$$

Los terrenos con mayor probabilidad de tener una transitabilidad en el rango $[0{,}5, 0{,}9]$ serán los más deseados.

3.4.4. Variables de Medida

Definimos la variable de medida y_j^k tal que $y_j^k = 1$ si $\mathrm{Tr}\left(c_{T,j}^k, \pi\right) \in [0{,}5, 0{,}9]$. Podemos reescribir la deseabilidad del terreno en términos de la probabilidad de la variable de medida.

$$\Pr\left(y_j^k \mid c_{T,j}^k\right) = \Pr\left(\mathrm{Tr}\left(c_{T,i}^k, \pi\right) \in [0{,}5, 0{,}9]\right) = \mathrm{Td}\left(c_{T,j}^k, \pi\right)$$

En la implementación, dicha probabilidad de medida se computa reemplazando la esperanza matemática por un promedio de datos obtenidos durante el entrenamiento:

$$\Pr\left(y_j^k \mid c_{T,j}^k\right) \approx \sum^{N_{\mathrm{traj}}} \frac{\mathbf{1}\left(\mathrm{Tr}\left(c_{T,j}^k, \pi\right) \in [0{,}5, 0{,}9]\right)}{N_{\mathrm{traj}}}$$

donde N_{traj} denota la cantidad de trayectorias obtenidas utilizando $c_{T,j}^k$. Notar que en el algoritmo las trayectorias también se utilizan para entrenar la política π.

Algorithm 1 Entrenamiento con currículum automático de terrenos

Inicializar la memoria de replay, muestrear $N_{particle}$ $c_{T,0}$s uniformemente de $\mathcal{C}$, $i, j = 0$.
while Todavía no hay convergencia **do**
  for $0 \leq k \leq N_{evaluate}$ **do**
    for $0 \leq l \leq N_{particle}$ **do**
      for $0 \leq m \leq N_{traj}$ **do**
        Generar un terreno usando $c_{T,j}^l$.
        Inicializar el robot.
        Correr la política π_i.
        Computar la traversibilidad para cada transición de estado.
        Guardar la trayectoria con sus valores de traversabilidad
      end for
    end for
    Actualizar la política con PPO.
    $i = i + 1$
  end for
  for $0 \leq l \leq N_{particle}$ **do**
    Computar la probabilidad de medición para cada parámetro $c_{T,j}^l$.
  end for
  for $0 \leq l \leq N_{particle}$ **do**
    Actualizar los pesos $\omega_j = \frac{P(y_i^l | c_{T,j}^l)}{\sum_m P(y_i^m | c_{T,j}^m)}$.
  end for
  Muestrear nuevamente $N_{particle}$ parámetros.
  Agregar los $c_{T,j}$s a la memoria de replay.
  for $0 \leq l \leq N_{particle}$ **do**
    Muestrear de la memoria de replay con probabilidad p_{replay}.
    Mover $c_{T,j}^l$ a un valor adyacente en $\mathcal{C}$ con probabilidad $p_t ransition$.
  end for
  $j = j + 1$
end while

En el algoritmo 1 utilizamos un filtro de partículas para almacenar una distribución discreta de terrenos deseables c_T durante el entrenamiento. Un filtro de partículas es un algoritmo utilizado para aproximar una distribución de probabilidad no paramétrica de forma iterativa. En este caso, nos interesa modelar la distribución de terrenos c_T tal que $\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]$.

Este algoritmo comienza con un conjunto de terrenos, también llamados partículas, aleatorios e iterativamente se actualizan dichas partículas hasta converger a la distribución objetivo.

Inicialmente, aproximamos la distribución con un conjunto finito de partículas, $c_T^k \in \mathcal{C}, k \in 1, \cdots, N_{\text{particle}}$. Estas partículas se inicializan de manera uniforme con rugosidad $r \in [0, 0{,}05]$, frecuencia $f \in [0{,}2, 1]$, y amplitud $a \in [0{,}2, 3]$ de forma tal que los terrenos iniciales sean lo suficientemente fáciles de transitar e incrementar la velocidad inicial del entrenamiento. El algoritmo se basa en las siguientes suposiciones:

- Los parámetros de terrenos con $\mathrm{Tr}\left(., \pi\right)$ similares, están cerca en Espacio Euclídeo.
- Una política π que fue entrenada en un área en C podrá interpolar a terrenos cercanos.

El ciclo principal selecciona una partícula c_T, y durante N_{Traj} iteraciones ejecuta la política π sobre el terreno generado. En cada una de estas iteraciones, se computa la transitabilidad del terreno definida anteriormente, y se actualizan los pesos de la política utilizando el algoritmo PPO.

Luego de repetir dicho procedimiento $N_{evaluate}$ veces, se computan las probabilidades de medida $\Pr\left(y_j^k \mid c_{T,j}^k\right)$ de manera empírica para cada una de las partículas.

Se normalizan las probabilidades de medida para obtener una nueva distribución de probabilidad discreta, de manera tal que la probabilidad de seleccionar la $k - esima$ partícula es w_k. De esta forma, se le asigna una probabilidad más alta a aquellas partículas tales que $\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]$. Finalmente, se realiza una nueva muestra utilizando los w_k y con probabilidad $p_{transition}$ se avanzan todos los parámetros de la partícula seleccionada mediante un *random walk* a una nueva partícula en su vecindario.

Basado en estas suposiciones, el filtro de partículas irá iterativamente incrementando la dificultad de los terrenos hasta converger a la distribución de c_T tal que $\mathrm{Tr}\left(c_T, \pi\right) \in [0{,}5, 0{,}9]$ y el agente se irá progresivamente adaptando a los nuevos terrenos generados asegurando un aprendizaje continuo.

3.5. Detalles de Implementación

En esta sección, describimos cómo los diferentes módulos interactúan entre sí para lograr el entrenamiento del robot hexápodo. Algunos módulos del sistema están implementados en Python y otros en C++, para garantizar la ejecución a 50Hzm, como se detalla a continuación:

- Phantomx.py: Módulo de Python que implementa la interfaz Gym y se utiliza para el entrenamiento mediante la biblioteca Stable Baselines.
- Server.cpp: Código C++ responsable de realizar un paso de simulación. Constantemente escucha y publica sobre tópicos de ROS. Es necesario para garantizar la ejecución a 50 Hz.

- Gait-Controller.cpp: Código C++ responsable de consumir datos de la red neuronal y mover las articulaciones del robot.

3.5.1. Phantomx.py

Este módulo de Python implementa la interfaz Gym vista anteriormente. Stable baselines utiliza este módulo para realizar entrenamiento. Por razones de eficiencia que discutiremos en breve, este módulo es simplemente un proxy que delega la implementación concreta a Server.cpp

3.5.2. Server.cpp

Es un módulo implementado en C++ que se encarga de implementar el método step: tomar una acción y retornar la siguiente observación en el tiempo. Se comunica con el módulo Phantomx.py mediante el uso de web sockets. La motivación detrás de la implementación en C se debe a:

- Alcanzar la frecuencia deseada, es decir publicar una nueva acción a 50HZ. Para lograr esto, debe ser eficiente la manera de pausar y despausar la simulación, sin desperdiciar ciclos de simulación en donde no se toman acciones. En el desarrollo de este trabajo, notamos que en Python estas operaciones tomaban demasiado tiempo, por lo cual fue necesario implementarlas a C++.
- En C++ contamos con la operación ros::spinOnce() para forzar que se publique el estado del robot en los tópicos de ROS en el momento que queramos. Esto nos permite retornar de inmediato la observación más reciente, sin desperdiciar ciclos de simulación.

La implementación del método step se presenta en detalle a continuación.

1. Durante el entrenamiento, la biblioteca Stable Baselines invocará al método step() en Phantomx.py, pasando como parámetro las acciones elegidas por la red neuronal.
2. Como mencionamos anteriormente, Phantomx.py es simplmente un Proxy, por lo que delega la implementación concreta al módulo Server.cpp. Para ello se comunica con dicho módulo mediante Web Sockets en un puerto específico. La información se codifica en un vector de bytes.
3. El módulo server.cpp recibe la invocación al método step y en primer lugar despausa la física del simulador utilizando el servicio Gazebo $/gazebo/unpause - physics$
4. Luego utiliza los tópicos de ROS para publicar acciones correspondientes a $delta_r$ y la frecuencia de cada pata en gait-controller.cpp. Este módulo utilizará cinemática inversa para calcular el ángulo de cada servo requerido para realizar el movimiento y pasar al siguiente estado en la simulación.
5. Usamos el comando $ros :: spinOnce()$, que solo está disponible en C++, para forzar la publicación de todos los tópicos de ROS en nuestro entorno que son necesarios para obtener el siguiente vector de observación.

6. Usamos el servicio Gazebo $/gazebo/pause-physics$ para pausar la física del simulador hasta que sea invocado nuevamente el método step.

7. Por último, leemos las siguientes observaciones usando el tópico ROS correspondiente y lo devolvemos a Phantomx.py para que pueda usarlo para predecir la siguiente acción.

3.5.3. Gait-Controller.cpp

Es el módulo encargado de mover físicamente el robot. Escucha los tópicos de ROS para obtener los $delta_r$ y la frecuencia de cada pata. Luego utiliza el modelo de cinemática inversa del robot para publicar los nuevos ángulos deseados para cada una de las articulaciones.

3.5.4. Listado de topicos

A continuación se detallan los distintos tópicos disponibles para interactuar con el simulador.

Tópicos del Sistema ROS y Gazebo

Tópico	Descripción
`/clock`	Proporciona el tiempo simulado en Gazebo.
`/gazebo/link_states`	Estado de los enlaces en la simulación de Gazebo.
`/gazebo/model_states`	Estado de los modelos en la simulación de Gazebo.
`/gazebo/parameter_descriptions`	Descripciones de los parámetros de Gazebo.
`/gazebo/parameter_updates`	Actualizaciones de los parámetros de Gazebo.
`/gazebo/performance_metrics`	Métricas de rendimiento de la simulación de Gazebo.
`/gazebo/set_link_state`	Permite establecer el estado de un enlace en Gazebo.
`/gazebo/set_model_state`	Permite establecer el estado de un modelo en Gazebo.
`/ground_truth/base_link/state`	La posición del robot en las coordenadas de su base.

Tópicos Específicos del Robot PhantomX

Tópico	Descripción
`/phantomx/base_frec`	Frecuencia base del robot PhantomX.
`/phantomx/delta_r/[pierna]`	Delta r para cada pierna del robot.
`/phantomx/frec/[pierna]`	Frecuencia para cada pierna del robot.

/phantomx/j_[articulación]_position_controller/command	Comando para el controlador de posición de cada articulación.
/phantomx/j_[articulación]_position_controller/state	Estado del controlador de posición para cada articulación.
/phantomx/phase/[pierna]	Fase de movimiento para cada pierna del robot.

Tópicos de Física y Contacto

Tópico	Descripción
`/physics/contacts_forces_[parte]`	Fuerzas de contacto para diferentes partes del robot.

Tópicos del Sistema ROS

Tópico	Descripción
`/rosout`	Salida de registro del sistema ROS.
`/rosout_agg`	Agregación de mensajes de registro de ROS.
`/tf`	Transformaciones entre marcos de coordenadas.
`/tf_static`	Transformaciones estáticas entre marcos de coordenadas.

Grafo de Nodos y Tópicos

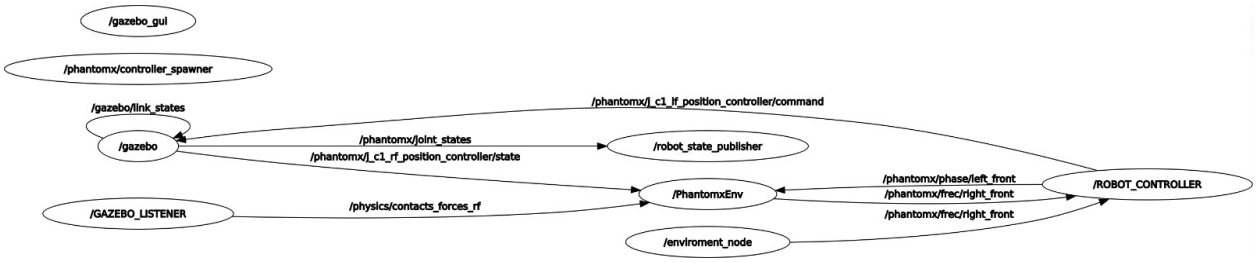

Fig. 3.15: Nodos y tópicos de ROS.

En la Figura 3.15 se muestra la estructura general de nuestra aplicación en ROS. Los nodos están representados por círculos y los tópicos se indican mediante arcos. La dirección de cada arco señala que un nodo envía datos a otro nodo a través de un tópico de ROS.

Primero, tenemos el nodo principal *PhantomXEnv*, que es el nodo de ROS encargado de interactuar con el entorno de Gazebo. Este nodo recibe información del robot a través de varios tópicos, como *joint states*, *position controller*, y *contact forces*, y utiliza estos datos para crear un vector que representa el estado actual del sistema. El nodo *PhantomXEnv* avanza un paso en el entorno y publica los resultados de la red neuronal, es decir, las frecuencias y fases de cada pata, al nodo *robot controller*. Este último se encarga de calcular la siguiente posición de cada pata y luego aplicar cinemática inversa para determinar las fuerzas necesarias en cada articulación. Finalmente, el nodo *robot controller* publica las fuerzas que se aplicarán en cada pata al nodo de Gazebo a través de los tópicos de comando.

3.5.5. Plugins de Gazebo

Un plugin en Gazebo es un archivo de código, generalmente con extensiones `.so` (shared objects), que se utiliza para extender la funcionalidad del simulador. Estos plugins permiten a los usuarios personalizar y controlar el comportamiento de los objetos y sensores en la simulación, añadiendo características específicas que no están disponibles de

manera predeterminada. Los plugins pueden estar escritos en C++ o Python y se integran directamente en el archivo URDF en donde se describe al robot. En general, los plugins exponen información mediante el uso de topicos de ROS.

El siguiente es un ejemplo del plugin `libgazebo_ros_p3d.so`, que tiene la función de publicar el estado de la base del robot, `base_link` en el tópico ROS `/ground_truth/base_link/state`.

```
<plugin name="p3d_base_controller" filename="libgazebo_ros_p3d.so">
  <alwaysOn>true</alwaysOn>
  <updateRate>200.0</updateRate>
  <bodyName>base_link</bodyName>
  <topicName>/ground_truth/base_link/state</topicName>
  <gaussianNoise>0.0</gaussianNoise>
  <frameName>world</frameName>
  <xyzOffsets>0 0 0</xyzOffsets>
  <rpyOffsets>0 0 0</rpyOffsets>
</plugin>
```

En el, se publicará información sobre la posición y la orientación del robot a una frecuencia de 200Hz.

Otro plugin utilizado es `gazebo_ros_control`, que nos expone 18 tópicos, uno por cada articulación del robot, para publicar el ángulo en el cual queremos posicionarlo.

```
 <gazebo>
    <plugin filename="libgazebo_ros_control.so" name="gazebo_ros_control">
      <robotNamespace>/phantomx</robotNamespace>
    </plugin>
  </gazebo>
```

Adicionalmente, en este trabajo construimos nuestro propio plugin `phantomx_heightmap_plugin` y lo utilizamos para computar las alturas de los puntos cercanos, utilizado en uno de los términos de la función de recompensa.

En particular, dada una posición (x, y) en el terreno, y un radio r, nos retorna las alturas de todos los puntos en el terreno en un circulo de radio r centrado en (x, y). Esto es útil para saber si nos estamos aproximando a un área del terreno con mayor elevación y así poder anticiparnos a mover las articulaciones de forma correcta.

4. RESULTADOS

En este capítulo presentamos los resultados de los experimentos que se realizaron para validar el sistema desarrollado que se encuentra disponible para la comunidad en `https://github.com/lrse/hexapod-locomotion/`. Estos son los mejores resultados que obtuvimos luego de probar con distintos conjuntos de hiperparámetros.

Los experimentos se corrieron en una computadora con las siguientes prestaciones: Intel i7-5600U, 2.6 – 3.2GHz, dual-core 64-bit, 16GB de memoria RAM y una tarjeta gráfica NVIDIA GeForce RTX 3070 Ti.

Un resultado importante previo a los experimentos fue la implementación del sistema a una frecuencia de muestreo de 50 Hz. Esta alta frecuencia de muestreo fue fundamental para permitir la ejecución rápida de acciones, esencial para la cinemática del robot hexápodo. La optimización se logró mediante una implementación híbrida Python-C, diseñada para acelerar las llamadas a funciones que mostraban una sobrecarga computacional significativa en entornos implementados en Python.

En el primer experimento, se entrenó el sistema en terreno plano. El resultado de dicho entrenamiento puede verse en la Figura 4.1, donde se muestra el refuerzo por paso de entrenamiento. Como material suplementario se presenta un video con un episodio de caminata con la política obtenida. La caminata resultante mantiene leves irregularidades en la forma de mover las patas, pero es aceptable y capaz de trasladarse en la dirección deseada.

Fig. 4.1: Curva de aprendizaje para terreno plano.

Una vez alcanzada una política para caminar en plano, se procedió a probar con y sin el curriculum de terrenos adaptativo. En el primer caso se entrenó con terrenos aleatorios con parámetros uniformemente distribuidos en todo el rango y en el segundo caso con el Filtro de Partículas. En la Figura 4.2 se pueden ver ambas curvas de aprendizaje, donde el resultado para el Filtro de Partículas se finalizó ni bien contamos con información suficiente para realizar la comparación, y la figura naranja, correspondiente al entrenamiento sin el filtro de partículas ya convergió. Se puede observar que el resultado con curriculum de terrenos adaptativo obtiene mayor recompensa final en un menor tiempo de entrenamiento,

y cualitativamente se observó que la política final obtenida es mejor.

Fig. 4.2: Curvas de aprendizaje utilizando el filtro de partículas para seleccionar los terrenos (celeste) y con terrenos aleatorios (azul).

En las Figuras 4.3 - 4.8 se analizan diversas métricas correspondientes a la política entrenada utilizando el filtro de partículas. En las figuras 4.3 y 4.4, se presentan la recompensa promedio y la duración promedio del episodio a medida que aumentan los pasos de entrenamiento. Se observa una tendencia positiva en ambas curvas, y esto se corresponde con el robot hexápodo mejorando su caminata consistentemente a lo largo del tiempo.

En la figura 4.5, se muestra la evolución de la función de pérdida asociada a la estimación del valor V. Se observa una disminución significativa durante las primeras iteraciones, una fase caracterizada por una mayor exploración, seguida de una rápida estabilización. Un valor bajo en la función de pérdida indica que el modelo realiza buenas predicciones sobre el valor V asociado a cada estado.

Las figuras 4.6 y 4.7 muestran la desviación estándar y la varianza explicada durante el entrenamiento. Una disminución en estos valores sugiere que el modelo está convergiendo hacia una solución óptima.

En particular, la varianza explicada [21] es una medida de qué tan bien las predicciones del modelo explican la variabilidad en los datos observados. Un valor más alto de varianza explicada indica que el modelo predice de manera más precisa los retornos esperados para los diferentes pares de estados y acciones.

Finalmente, en la figura 4.8 se grafica la componente ponderada de la función de recompensa encargada de hacer que el robot siga el comando de velocidad especificado, maximizando el producto interno. A medida que el tiempo de entrenamiento progresa se observa un incremento en este término de la función de recompensa, estabilizándose en alrededor de la mitad de las iteraciones, lo cual confirma nuestras observaciones en simulador de que el robot sigue de manera más efectiva las instrucciones proporcionadas.

Confirmada la utilidad del currículo de terrenos, se procedió a generar terrenos irregulares para comprobar la adaptabilidad de esta política. En el material suplementario se presentan videos con ejemplos de caminata con la política obtenida. Se observa que el robot logra aprender a caminar en los terrenos irregulares generados.

Tanto en terreno plano como en terrenos irregulares podemos observar que el robot alcanza una trayectoria subóptima, pero suficiente para poder transitar, con un movimiento

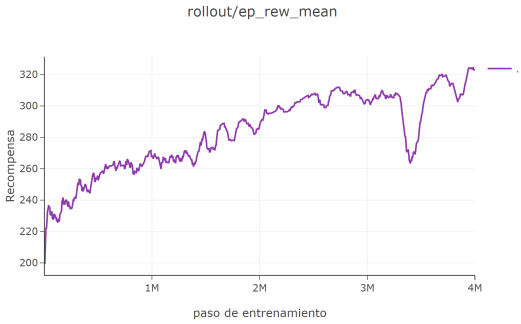

Fig. 4.3: Curvas de aprendizaje utilizando el filtro de partículas. El eje horizontal representa la cantidad de pasos y el eje vertical la recompensa promedio.

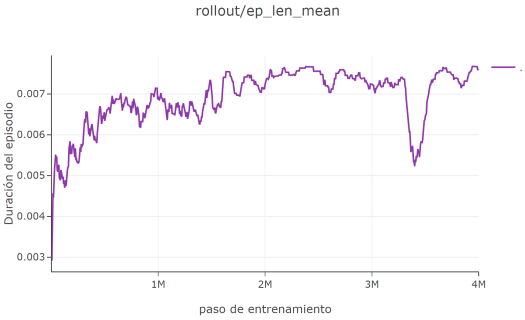

Fig. 4.4: Curvas de aprendizaje utilizando el filtro de partículas. El eje horizontal representa la cantidad de pasos y el eje vertical la duración promedia del episodio

de patas razonable, aunque no simétrico, ya que no hay nada en el esquema de aprendizaje por refuerzo ni en la función de recompensa utilizada que contemple la simetría del robot. Si observamos por varios segundos cómo el robot responde al comando de velocidad deseada, veremos que eventualmente se desvía de la dirección original, pero esto se debe a que el comando de velocidad se da a lazo abierto para el sistema global, es decir, ante una perturbación que corra al robot de su camino en una dirección éste continuará hacia adelante sin corregir que su postura se desvió de la orientación inicial. En un sistema que utilizara la política obtenida esto lo podría corregir el teleoperador o el sistema de planificación de trayectoria que sí tendrían disponible información de la posición global del robot.

En primer lugar, el entrenamiento en terreno plano demostró la capacidad del sistema para aprender una política de caminata funcional. Aunque la caminata resultante mantuvo leves irregularidades en el movimiento de las patas, logró el objetivo principal de trasladarse en la dirección deseada.

Un hallazgo particularmente significativo fue la comparación entre el entrenamiento con terrenos aleatorios y el uso de un currículo adaptativo basado en el Filtro de Partículas. Los resultados muestran que el currículo adaptativo no solo acelera el aprendizaje, sino que también resulta en una política final capaz de generalizar a una variedad de terrenos

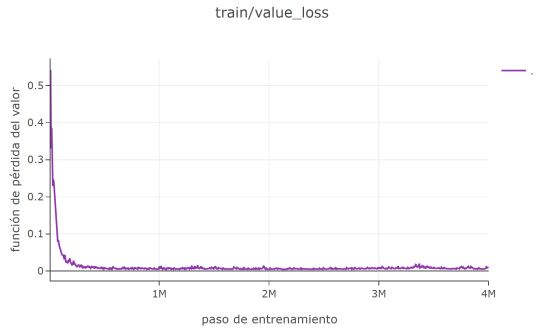

Fig. 4.5: El eje horizontal representa la cantidad de iteraciones y el eje vertical la función de pérdida del valor V.

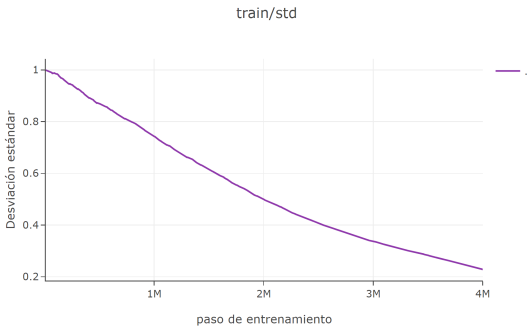

Fig. 4.6: El eje horizontal representa la cantidad de iteraciones y el eje vertical la desviación estándar durante el entrenamiento.

irregulares. Esto es particularmente importante para aplicaciones en el mundo real, donde los robots deben enfrentarse a entornos desconocidos y cambiantes.

Una de las limitaciones más significativas de este estudio fue la disponibilidad de recursos computacionales. Aunque contábamos con GPUs de alto rendimiento, que fueron cruciales para la aceleración de ciertos cálculos, nos encontramos con una escasez de CPUs. Esta limitación restringió nuestra capacidad para realizar experimentos más exhaustivos y explorar un espectro más amplio de hiperparámetros.

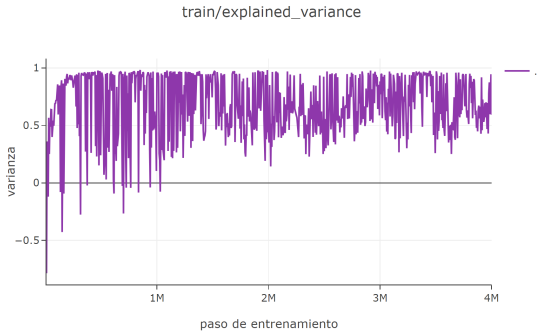

Fig. 4.7: El eje horizontal representa la cantidad de iteraciones y el eje vertical la varianza explicada durante el entrenamiento.

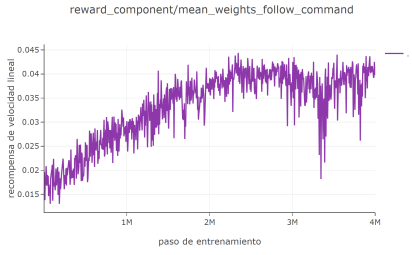

Fig. 4.8: El eje horizontal representa la cantidad de iteraciones y el eje vertical el componente de la función de recompensa responsable de seguir la dirección del comando.

5. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo se presentó un sistema de locomoción para robot hexápodo basado en aprendizaje por refuerzo profundo en simulación utilizando únicamente mediciones propiocépticas. El sistema obtenido pudo aprender a caminar en terrenos irregulares; la caminata final presenta ciertas limitaciones, pero logra transitar todo el terreno a pesar de ellas.

Una de las mayores dificultades encontradas durante la realización de este trabajo fue la demanda temporal del simulador para el entrenamiento. A pesar de las simplificaciones realizadas al sistema, el entrenamiento para un único comando de velocidad demanda alrededor de una semana en una computadora con las siguientes prestaciones: Intel i7-5600U, 2.6 – 3.2GHz, dual-core 64-bit, 16GB de memoria RAM y una tarjeta gráfica NVIDIA GeForce RTX 3070 Ti. Esto se debe a que no se pudieron obtener velocidades de simulación mayores a real time en Gazebo Classic, y al requerirse millones de episodios para el entrenamiento el tiempo real escala a valores que impiden realizar pruebas eficientes, ajustar mejor los hiperparámetros y/o entrenar con más comandos de velocidad. Por lo tanto, una de las líneas de trabajo futuro consiste en explorar otras plataformas de simulación para el entrenamiento, que permitan sacar provecho de las tarjetas gráficas.

En cuanto al curriculum de terrenos adaptativo, se planea incorporar terrenos de tipo escalera, así como ajustar más los parámetros de los terrenos generados para obtener una variedad más acorde a los tipos de terrenos que se encontrará la política obtenida en su aplicación final. Por otro lado, como se mencionó previamente, la función de recompensa no contempla explícitamente la simetría del robot, ni tampoco la estabilidad estática y/o dinámica del mismo; estas también son cuestiones a abordar en un futuro.

En cuanto a las herramientas de simulación utilizadas en este estudio, es importante señalar que, si bien Ros y Gazebo han sido fundamentales para nuestro trabajo actual, reconocemos las limitaciones que estas plataformas presentan para tareas de aprendizaje por refuerzo a gran escala. En este sentido, una dirección futura crucial para nuestra investigación es la transición planificada a Isaac Sim, de Nvidia. Esta plataforma de simulación avanzada está específicamente diseñada para ser más adecuada para tareas de aprendizaje por refuerzo, ofreciendo mejoras significativas en términos de rendimiento, escalabilidad y fidelidad de simulación.

Al utilizar frameworks específicamente diseñados para aprendizaje por refuerzo, se podría explorar el uso de la librería Optuna [22], un framework de optimización de hiperparámetros, para mejorar el rendimiento de los modelos. Esto nos permitiría realizar una búsqueda eficiente de hiperparámetros, lo que lograría mejorar aún más la convergencia y estabilidad de los algoritmos.

El presente trabajo se enmarca en un objetivo más ambicioso que consiste en implementar un sistema de locomoción basado en aprendizaje maestro-estudiante en un robot hexápodo. Los resultados obtenidos presentan un primer paso en esta dirección. Una vez desarrollado el sistema completo maestro-estudiante, se evaluará la política obtenida con el estudiante en el robot real PhantomX Mark II.

Bibliografía

[1] J. L. Yulong Lu, "A universal approximation theorem of deep neural networks for expressing probability distributions," in *Thirty-fourth Conference on Neural Information Processing Systems, NeurIPS*, 2020.

[2] D. A. W. D. Michael Bowling, John D. Martin, "Settling the reward hypothesis," in *Settling the Reward Hypothesis*, 2020.

[3] A. G. B. Richard S. Sutton, *Reinforcement Learning: An Introduction.* The MIT Press, 2015.

[4] J. Achiam, "Spinning Up in Deep Reinforcement Learning," *Spinning Up OpenAI*, 2018. [Online]. Available: https://spinningup.openai.com

[5] M. P. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, 2011.

[6] e. a. David Silver, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint*, 2017.

[7] T. Li, N. Lambert, R. Calandra, F. Meier, and A. Rai, "Learning generalizable locomotion skills with hierarchical reinforcement learning," in *2020 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, 2020, pp. 413–419.

[8] W. Ouyang, H. Chi, J. Pang, W. Liang, and Q. Ren, "Adaptive locomotion control of a hexapod robot via bio-inspired learning," *Frontiers in Neurorobotics*, vol. 15, p. 627157, 2021.

[9] M. Schilling, K. Konen, F. W. Ohl, and T. Korthals, "Decentralized deep reinforcement learning for a distributed and adaptive locomotion controller of a hexapod robot," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, 2020, pp. 5335–5342.

[10] T. Azayev and K. Zimmerman, "Blind hexapod locomotion in complex terrain with gait adaptation using deep reinforcement learning and classification," *Journal of Intelligent & Robotic Systems*, vol. 99, no. 3, pp. 659–671, 2020.

[11] G. L. F. R. Joana Coelho, Bruno Dias and P. Flores, "Development and implementation of a new approach for posture control of a hexapod robot to walk in irregular terrains," *Cambridge University Press*, 2021.

[12] A. M. M. S. Kai Konen, Timo Korthals, "Biologically-inspired deep reinforcement learning of modular control for a six-legged robot," *Cluster of Excellence Cognitive Interaction Technology*, 2022.

[13] A. Agarwal, A. Kumar, J. Malik, and D. Pathak, "Legged locomotion in challenging terrains using egocentric vision," in *Conference on robot learning.* PMLR, 2023, pp. 403–415.

[14] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning robust perceptive locomotion for quadrupedal robots in the wild," *Science Robotics*, vol. 7, no. 62, p. eabk2822, 2022.

[15] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain," *Science robotics*, vol. 5, no. 47, p. eabc5986, 2020.

[16] D. H. K. G. P. A. Philipp Wu, Alejandro Escontrela, "Daydreamer: World models for physical robot learning," *Proceedings of Machine Learning Research*, 2022.

[17] Gazebo. [Online]. Available: https://gazebosim.org/

[18] Ros. [Online]. Available: https://www.ros.org/

[19] Stable baselines 3. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/

[20] Phantomx model. [Online]. Available: https://github.com/HumaRobotics/phantomx_description

[21] Scikit-learn Developers. (2024) Scikit-learn Model Evaluation: Explained variance score. [Online]. Available: https://scikit-learn.org/stable/modules/model_evaluation.html#explained-variance-score

[22] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.