Universidad Zaragoza

Trabajo Fin de Grado

Evaluación de algoritmos de aprendizaje por refuerzo profundo con un motor físico

Evaluation of Deep Reinforcement Learning algorithms with a physics engine

Autor

Francisco Navarro Soler

Director

Rubén Martínez Cantín

Grado en Ingeniería de Tecnologías Industriales

Escuela de Ingeniería y Arquitectura

2021

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de Estudios de la titulación de (Título del Trabajo)

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, a 23 de Septiembre de 2021

Fdo: Francisco Navarro Soler

RESUMEN

Uno de los grandes retos actuales de la robótica es poder explorar y navegar por entornos totalmente desconocidos y en muchas ocasiones extremos. Para ello es de vital importancia que los robots sean capaces de aprender, adaptarse y recuperarse de los errores de la manera más rápida y eficiente posible.

El aprendizaje por refuerzo es una rama del aprendizaje automático que se encarga de que las máquinas aprendan a desenvolverse a base de premios y castigos, en entornos no controlados de los cuales no tenemos información inicial.

El principal objetivo de este TFG es estudiar y analizar un conjunto de algoritmos Actor-Critic implementados en Stable Baselines 3 mediante el motor físico de Pybullet.
La utilización de algoritmos complejos que emplean aprendizaje del doble valor Q, la actualización de la política con retraso, el suavizado de la política objetivo y la regularización de entropía nos ha permitido obtener un mejor entrenamiento y comportamiento del agente.

Se ha evaluado el entorno CartPole de OpenAI Gym durante 100.000 timesteps, el CartPole de Pybullet durante 40.000 timesteps, el HalfCheetah, Hopper y el Ant de Pybullet durante 200.000 timesteps y por último otra vez el Ant para 668.000 timesteps. Para que los resultados fueran robustos y fiables se realizaron cuatros entrenamientos en el CartPole y tres en el Halfcheetah, en el Hopper y en el Ant respectivamente.

ABSTRACT

Nowadays one of the great challenges of robotics is to be able to explore and navigate in totally unknown and often extreme environments. To this end, it is vitally important for robots to be able to learn, adapt and recover from errors as quickly and efficiently as possible.

Reinforcement learning is a branch of machine learning that teaches machines to learn to behave based on rewards and punishments, in uncontrolled environments of which we have no initial information.

The main objective of this Final Degree Project is to study and analyze a set of Actor-Critic algorithms implemented in Stable Baselines 3 using the Pybullet physics engine.
The use of complex algorithms employing double Q-value learning, delayed policy update, target policy smoothing and entropy regularization has allowed us to obtain better agent training and behavior.

We evaluated OpenAI Gym's CartPole environment for 100,000 timesteps, Pybullet's CartPole for 40,000 timesteps, Pybullet's HalfCheetah, Hopper and Ant for 200,000 timesteps and finally Ant for 668.000 time steps. To make the results robust and reliable, four training runs were performed on the Cartpole and three on each Halfcheetah, Hopper and Ant environments respectively.

ÍNDICE

RESUMEN ..... - 0 -

ABSTRACT ..... - 1 -

1. INTRODUCCIÓN ..... - 4 -

1.1. OBJETO ..... - 4 -
1.2. ALCANCE ..... - 5 -
1.3. METODOLOGÍA ..... - 5 -

2. APRENDIZAJE POR REFUERZO ..... - 7 -

2.1. ELEMENTOS EN EL APRENDIZAJE POR REFUERZO ..... - 8 -
2.2. MÉTODOS DEL APRENDIZAJE POR REFUERZO. ..... - 9 -
2.3. ALGORITMOS ACTOR-CRITIC ..... - 9 -
2.3.1. BÚSQUEDA DE POLÍTICA ..... *- 10 -*
2.3.2. ACTOR: BÚSQUEDA DE POLÍTICA CON FUNCIONES VALOR ..... *- 11 -*
2.3.3. ESTIMANDO Q CON REDES NEURONALES ..... *- 12 -*
2.4. ALGORITMOS EMPLEADOS ..... - 13 -
2.4.1. SAC ..... *- 13 -*

3. ENTRENAMIENTOS Y RESULTADOS ..... - 14 -

3.1. ENTORNOS ..... - 15 -
3.2. CARTPOLE DE OPEN AI GYM ..... - 18 -
3.3. CARTPOLE DE PYBULLET ..... - 19 -
3.3.1. A2C ..... *- 20 -*
3.3.2. PPO ..... *- 21 -*
3.3.3. TD3 ..... *- 22 -*
3.3.4. SAC ..... *- 22 -*
3.4. HALFCHEETAH, HOPPER Y ANT ..... - 25 -

CONCLUSIONES ..... - 28 -

BIBLIOGRAFÍA ..... - 29 -

A. ANEXO A: ..... - 30 -

A. PROCESOS DE DECISIÓN DE MARKOV ..... - 30 -
i. Proceso de decisión ..... *- 30 -*
ii. Política ..... *- 31 -*
iii. Utilidad ..... *- 31 -*
iv. Función de valor y ecuaciones de Bellman ..... *- 31 -*
v. Funciones de valor óptimas y política ..... *- 32 -*
B. HERRAMIENTAS DE PREDICCIÓN ..... - 33 -
i. Evaluación de Montecarlo ..... *- 33 -*
ii. Diferencia Temporal ..... *- 34 -*
C. OPTIMIZACIÓN DE LA POLÍTICA. ..... - 35 -
i. Basado en la función de valor ..... *- 35 -*
ii. Basado en la búsqueda de la política ..... *- 36 -*

ÍNDICE DE FIGURAS

Figura 1. Ilustración del entorno en el aprendizaje por refuerzo - 8 -
Figura 2. Esquema de algoritmos de aprendizaje por refuerzo - 9 -
Figura 3. Esquema de los algoritmos Actor-Critic - 10 -
Figura 4. Stochastic Gradient Descent (SGD) - 11 -
Figura 5. Entorno CartPole de OpenAI - 15 -
Figura 6. Entorno CartPole de Pybullet - 16 -
Figura 7. Entorno de Halfcheetah de Pybullet - 17 -
Figura 8. Entorno de Hopper de Pybullet - 17 -
Figura 9. Entorno de Ant de Pybullet - 18 -
Figura 10. Evaluación de CartPole-v0 Figura 11. Evaluación de CartPole-v1 - 19 -
Figura 12. Cuatro evaluaciones del algoritmo A2C - 20 -
Figura 13. Evaluación del entrenamiento de 100.000 time steps - 21 -
Figura 14. Cuatro evaluaciones del Algoritmo PPO - 21 -
Figura 15. Cuatro evaluaciones del algoritmo TD3 - 22 -
Figura 16. Cuatro evaluaciones del algoritmo SAC - 23 -
Figura 17. Evaluación de los cuatro algoritmos - 23 -
Figura 18. Evaluación de los cuatro algoritmos con ruido - 24 -
Figura 19. Evaluación del entrenamiento con SAC - 26 -
Figura 20. Evaluación del entrenamiento con SAC - 26 -
Figura 21. Evaluación del entrenamiento con SAC - 27 -
Figura 22. Renderizado de la evaluación de 'AntBulletEnv-v0' - 27 -

1. INTRODUCCIÓN

En la actualidad el aprendizaje automático está experimentando grandes avances. Esto ha sido posible gracias a los estudios académicos realizados en la última década y a las nuevas tecnologías que permiten procesar una gran cantidad de datos, necesarios para el buen entrenamiento y funcionamiento de los algoritmos de aprendizaje automático [1] .
Dentro del aprendizaje automático encontramos:

- Aprendizaje supervisado: Consiste en aprender, a partir de datos ya conocidos y etiquetados, cómo actuar en situaciones futuras parecidas.
- Aprendizaje no supervisado: Consiste en aprender estructuras ocultas en conjuntos de datos no etiquetados.
- Aprendizaje por refuerzo: Consiste en que el agente aprenda mediante interacción con el entorno a realizar las tareas encomendadas. El entrenamiento se realiza en base a recompensas y penalizaciones.

Este trabajo es un estudio de algoritmos de aprendizaje por refuerzo profundo por la necesidad de ampliar el conocimiento propio sobre los nuevos métodos de la inteligencia artificial que están siendo una revolución en muchos ámbitos, como por ejemplo la investigación, la robótica o la economía.
Está habiendo avances muy grandes en el sector de la conducción autónoma. Hay algoritmos que consiguen por ejemplo que los drones aprendan a hacer piruetas en el aire en cuestión de segundos cuando a un humano le costaría semanas aprender.
El aprendizaje por refuerzo está también muy presente en los videojuegos pudiendo aprender a jugar en horas a juegos realmente complejos. Hay algoritmos que han sido capaces incluso de vencer a los mejores jugadores de Go [2]. Además, el motor físico empleado en este trabajo es el que emplean muchas empresas para el desarrollo de videojuegos o la simulación de escenas en las películas lo cual nos muestra el potencial que tiene este motor para simular entornos reales.
El trading, que es otro tema de actualidad y que interesa mucho, es un sector en el que también se está empleando el aprendizaje por refuerzo para realizar inversiones a gran velocidad incluso con mejor criterio que los humanos.
Es por todo esto que el aprendizaje por refuerzo se está desarrollando tanto y seguirá evolucionando para satisfacer los deseos humanos de emplear robots que faciliten el trabajo diario y proporcionen beneficios.

1.1. OBJETO

En este Trabajo Fin de Grado (TFG) se emplea un simulador con un motor físico para la evaluación y estudio de algoritmos de aprendizaje por refuerzo profundo.
Los objetivos principales de este trabajo son dos:

- Realizar una comparación de algoritmos Actor-Critic y sus prestaciones para diferentes aplicaciones.
- Estudio de la influencia que tienen las diferentes formas de implementar los algoritmos en la evaluación de los modelos entrenados.

La evaluación se va a realizar en diferentes entornos de simulación para representar las diferentes tareas caminar y correr.

1.2. ALCANCE

El proceso seguido durante la realización del trabajo y el nivel de profundización en cada una de las actividades llevadas a cabo fue el siguiente. Primero se realizó un estudio acerca del aprendizaje automático en general, incidiendo posteriormente en el aprendizaje por refuerzo. En segundo lugar, se instaló Python y se trabajó con el entorno de desarrollo de Google Colab. Además, se hicieron pruebas para aprender a usar los algoritmos de Stable Baselines 3 [3] y las herramientas que estas implementaciones proporcionan. Una vez entendidas y controladas todas las herramientas se procedió a plantear los entornos y las tareas que se iban a realizar. Se realizaron experimentos con sistemas sencillos empleando cuatro algoritmos Actor-Critic (A2C, PPO, TD3 y SAC) para ver cuál de ellos aprendía y funcionaba mejor con el entorno y posteriormente emplear el algoritmo que mejor comportamiento presentara para realizar tareas más complejas. Tras tener claro las tareas y el sistema de trabajo se procedió a la evaluación de los cuatro algoritmos y la selección del mejor de ellos para realizar las tareas más complejas. Se utilizaron para ello los mismos programas y herramientas que en las pruebas de familiarización. Por último, con los resultados obtenidos se ha comparado el comportamiento de cada algoritmo con los entornos estudiados.

La cronología y duración de cada etapa ha sido la siguiente:
-Estudio del aprendizaje automático y por refuerzo, Python, Pybullet (2 semanas).
-Instalación y familiarización con los programas y herramientas a utilizar (2 semanas).
-Planteamiento de entornos y tareas (1 semana).
-Evaluación de algoritmos de aprendizaje por refuerzo (2 semanas).
-Comparativa de resultados (2 semanas).
-Documentación y escritura de la memoria (1 semana).

1.3. METODOLOGÍA

La parte de programación se ha llevado a cabo en Python y se ha trabajado principalmente en la nube con Google Colab. Esto se eligió así por comodidad ya que trabajando en la nube no era necesario la instalación en el ordenador de librerías ni programas adicionales como Anaconda. Además, se ofrece gratuitamente el acceso a una GPU que permite acelerar el proceso de entrenamiento de los modelos, lo cual sería imposible de usar con Python de manera local si no se dispone de un ordenador con GPU. Como primero se realizó el entrenamiento de los modelos y una vez guardados se analizaron los resultados, fue necesario mirar la documentación de Google Colab para aprender a guardar y descargar

documentos en carpeta .zip, así como subirlos a la nube y acceder a ellos para la realización de las gráficas. Las herramientas para realizar esto van incorporadas en Google Colab por lo que no fue necesario su instalación. El código de las evaluaciones se publicará en Google Colab o Github.
https://drive.google.com/drive/folders/136EmUmyVFD32CqZT4BeRSoc8h964yxgj?usp=sharing
El motor físico usado es Pybullet [4] con varios entornos basados en OpenAI Gym [5]. Pybullet es un módulo de Python basado en el motor de física Bullet Physics que sirve para la simulación de la física en la robótica, en los juegos, en los efectos visuales y en el aprendizaje automático ya que se pueden cargar cuerpos articulados desde URDF, SDF, MJCF y otros formatos de archivo. Pybullet proporciona simulación de dinámica directa, cálculo de dinámica inversa, cinemática directa e inversa, detección de colisiones y consultas de intersección de rayos [6]. Los motores de física como este son muy útiles para hacer una representación exacta del mundo real mediante software. Para saber qué entornos ofrece Pybullet y cómo hacer uso de ellos ha sido necesario seguir la documentación disponible en Github y el tutorial de iniciación para aprendizaje por refuerzo que se ofrecen.
Los algoritmos de aprendizaje por refuerzo usan el "framework" de Deep Learning Pytorch y la librería Stable Baselines 3. El Deep Learning es un conjunto de algoritmos de aprendizaje automático que se basa en la utilización de redes neuronales para realizar predicciones.
Stable Baselines 3 es un conjunto de implementaciones de aprendizaje por refuerzo en Pytorch. Hay que señalar que se comenzó usando la librería de Stable Baselines [7] porque era la que estaba disponible, pero durante la realización del trabajo se sacó una versión nueva llamada Stable Baselines 3 que es la que se acabó empleando. Esta versión está más actualizada y utiliza Pytorch en vez de Tensor Flow lo que se traduce en un código mucho más fácil de leer. Pytorch es un paquete de Python diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores. Además, también se ofrecía en esta nueva versión la posibilidad de trabajar con Tensorboard. Durante el transcurso del trabajo ha sido necesario hacer los tutoriales y correr el código de prueba que se ofrece Stable Baselines 3 así como leer y comprender la documentación disponible en Github centrándose en los algoritmos de estudio A2C, PPO, TD3 y SAC.

Dentro de la documentación se han estudiado en profundidad las secciones para:
- Crear los entornos y vectorizarlos para un mejor entrenamiento.
- Emplear los monitores para guardar información del entrenamiento.
- Emplear entornos de evaluación.
- Guardar los modelos.
- Emplear Callbacks para terminar el entrenamiento automáticamente o guardar los datos interesantes para poder verlos luego con Tensorboard

Por último, mencionar que para realizar las gráficas de los parámetros a evaluar durante el entrenamiento se ha empleado Tensorboard y Matplotlib. Tensor Board es una herramienta que permite monitorizar métricas como la recompensa o la función de pérdida durante el entrenamiento de modelos profundos y durante la evaluación y/o testeo. Permite también visualizar el grafo del modelo, lo cual es de gran ayuda para ver qué proceso está se está siguiendo. Matplotlib es una biblioteca que nos permite generar gráficos a partir de datos contenidos en listas o en "arrays" para el lenguaje de programación de Python.

Las gráficas que se muestran con Tensorboard están predeterminadas y no es posible cambiar los colores, ni añadir leyendas ni muchas otras funcionalidades por lo que fue necesario emplear código externo para acceder a los datos guardados. Estos se almacenaron con Numpy en forma de vector y posteriormente con Matplotlib se ajustaron las gráficas a nuestras conveniencias. Ha sido necesario leer la documentación disponible en internet para saber cómo trabajar con Numpy [8] y con Matplotlib [9]. Se aprendió a cómo obtener los datos de las tablas y almacenarlos en vectores, calcular la media y la desviación típica y a graficar estos resultados con la función de "error bar".
Se necesita instalar diferentes librerías y funciones para hacer uso de todas las herramientas mencionadas. Se estudió qué eran y cómo se instalaban mirando en las respectivas documentaciones de Stable Baselines 3, de Pybullet y de Tensorboard. Fue necesario un tiempo de adaptación y pruebas hasta que finalmente se dominó esta parte esencial para el desarrollo del trabajo.

2. APRENDIZAJE POR REFUERZO

El aprendizaje por refuerzo es una rama del aprendizaje automático que se encarga de que las máquinas aprendan a desarrollar su trabajo a base de premios y castigos en entornos no controlados, de los cuales no tenemos ninguna información inicial.
Los procesos de decisión de Markov (MDP) son descripciones formales del entorno de aprendizaje por refuerzo, como se explica en el anexo A. También se introducen los MDP como una tupla de la forma $\{\chi , A, T, R, \gamma\}$ donde tenemos:
Un conjunto de estados $x \in \chi$
Un conjunto de acciones $a \in A$
Una función de transiciones $T(x, a, x') = p(x'|x, a)$
Una función de recompensas $R(x, a, x')$
Al no tener ninguna información inicial y ser un entorno no controlado, de la tupla no conoceremos la función de transiciones T, ni la función de recompensa R, por lo que necesitaremos aprender estas funciones mediante "prueba y error" y experimentando.
Necesitamos que el agente aprenda a determinar la política óptima a base de interactuar con el entorno.
El "problema" típico del aprendizaje por refuerzo es equilibrar la explotación y la exploración. Explotar es tomar acciones que sabemos que son buenas gracias al entrenamiento. Explorar es hacer que el agente tome acciones nuevas para aprender, aunque no sepamos si estas acciones van a ser buenas a priori. Generalmente al explorar recibiremos recompensas notablemente peores por eso necesitamos un balance entre exploración y explotación.

2.1. ELEMENTOS EN EL APRENDIZAJE POR REFUERZO

Encontramos cuatro conceptos principales en el aprendizaje por refuerzo que son el agente, el entorno, la política y la recompensa. Tenemos que tener en cuenta que el entorno se modela como un proceso de decisión de Markov (MDP) en el que un agente percibe el estado de su entorno y debe tomar acciones que afecten a ese estado. Este estado sólo depende del estado anterior y no tendremos en cuenta el pasado lejano.
El agente es el modelo que queremos entrenar, éste se encuentra en un estado (s) y en cada interacción se obtiene una recompensa (r) y se toma una acción (a). El estado (s) es la ubicación dentro del entorno y la acción (a) es el desplazamiento a un nuevo estado (s').
Este agente "vive" en un mundo con el que interactúa que denominaremos el entorno que generalmente durante el entrenamiento es una simulación que tratará de ajustarse lo máximo posible al mundo real (Figura 1).

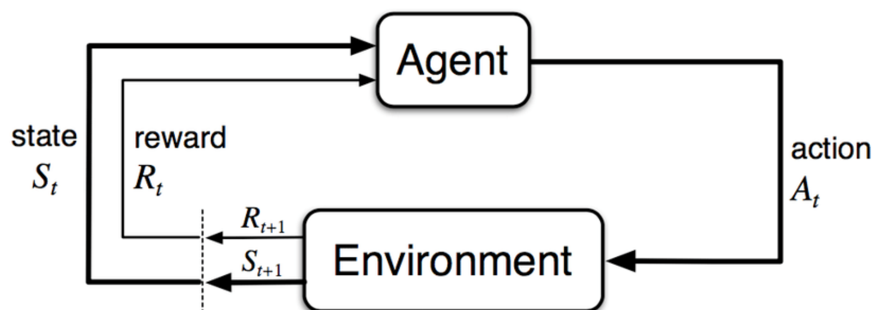


Figura 1. Ilustración del entorno en el aprendizaje por refuerzo

Es necesario que el agente siga una política, es decir unas reglas que le indiquen qué decisión tomar. Es muy importante aprender una buena política para desarrollar bien la tarea.
Como ya hemos dicho el agente obtiene una recompensa inmediata por cada interacción con el entorno. La recompensa es uno de los conceptos más importantes ya que indica si el agente está actuando correctamente en cada acción qué toma. El objetivo a largo plazo es maximizar la recompensa ya que esto indicará que el modelo está bien entrenado.
Es importante definir también unas recompensas "a largo plazo" y para ello definimos las funciones de valor las cuales estiman cuán positivo será para el agente llegar a un estado, teniendo en cuenta todas las acciones futuras posibles.
Dispondremos de dos funciones: La función de valor V que es la recompensa esperada para el agente al estar en un estado (s) y seguir una política (π). La función de acción-valor Q que es la recompensa esperada para el agente al estar en un estado (s), toma una acción (a) y posteriormente seguir la política (π). La función de valor óptima será la máxima función de valor entre todas las políticas.
Por último, aunque para tareas robóticas generalmente los espacios de actuación (conjunto de acciones válidas en un entorno) son continuos también hay algunas tareas más simples que pueden modelarse mediante tablas, es decir tienen un espacio de actuación discreto.
Los espacios continuos se modelarán mediante redes neuronales.

2.2. MÉTODOS DEL APRENDIZAJE POR REFUERZO.

Tenemos por un lado los algoritmos basados en un modelo donde se aprende un modelo aproximado basado en experiencias y se resuelve como un MDP conocido, asumiendo que el modelo es correcto. Su principal ventaja es su alta eficiencia. En cualquier caso, éstos no son objeto de estudio en este trabajo. Los algoritmos sin modelo son los que se van a analizar y comparar. Estos aprenden la política y las funciones de valor directamente por interacción. Son más fáciles de implementar y de ajustar sus parámetros y además son también los más estudiados y empleados en el ámbito de la robótica. Como no disponen de un modelo tienen que aprender la política óptima mediante la "prueba y error" [10].
Los modelos pueden optimizar la política de dos maneras (Figura 2):
- Hacer directamente una búsqueda de la política óptima. (Policy Optimization)
-Tratar de optimizar la política a partir de la función de valor. (Q-Learning)

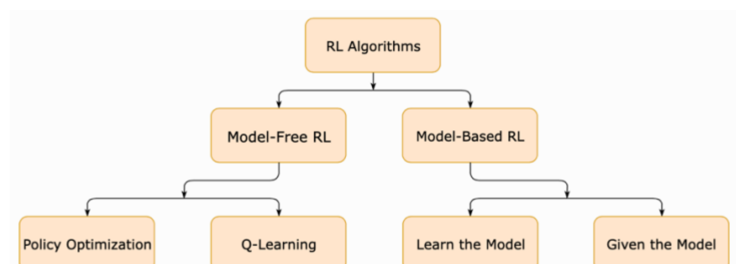


Figura 2. Esquema de algoritmos de aprendizaje por refuerzo

El estudio de este TFG se centra en los algoritmos Actor-Critic: A2C, PPO, TD3 y SAC. El PPO es solo Policy Optimization y los otros tres se encuentran entre Policy Optimization y Q-Learning ya que estiman una función Q para ayudar a la búsqueda de política.

2.3. ALGORITMOS ACTOR-CRITIC

Los algoritmos Actor-Critic, tienen dos series de parámetros, es decir dos redes neuronales que trabajan independientemente o una red neuronal con salidas independientes [11].
El Critic actualizará los parámetros φ de la función de acción-valor Q y el Actor actualizará los parámetros θ de la política en la dirección sugerida por el Critic. Se podría decir que el Critic se encarga de la evaluación de la política, mientras que el Actor se encarga de intentar tomar los mejores pasos (Figura 3). Hay que tener presente que en este algoritmo se emplea el TD error, el cual nos proporciona la diferencia entre la estimación actual del agente y el valor objetivo.

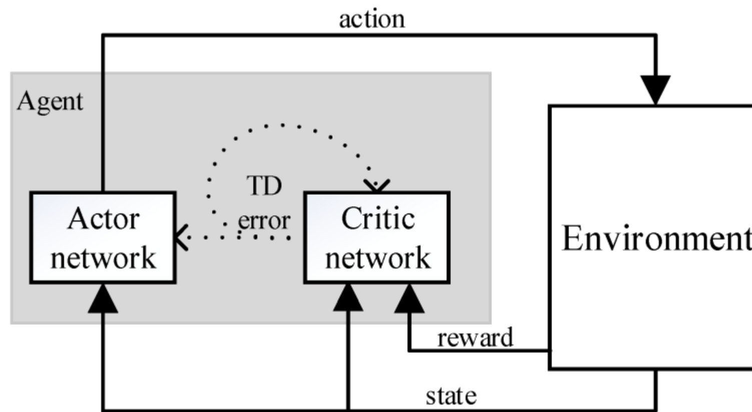


Figura 3. Esquema de los algoritmos Actor-Critic

2.3.1. BÚSQUEDA DE POLÍTICA

En el método de búsqueda de la política (Policy Search) en vez de emplear el valor Q se va a abordar directamente el problema de optimizar la política $\pi(x,a)$.
Se aprenden los parámetros de la función directamente mediante la interacción.
Para cualquier función $f(\cdot)$: $E_x[f(x)] = \sum_x \quad f(x) \cdot p(x)$

$$\nabla_\theta \, log f(\theta) = \frac{\nabla_\theta \, f(\theta)}{f(\theta)} \Rightarrow \nabla_\theta \, f(\theta) = f(\theta) \cdot \nabla_\theta \, log f(\theta)$$

Dada una política $\pi_\theta(x,a)$ se aprenden los parámetros de las funciones.
Para cuantificar cómo de buenos son los parámetros necesitamos definir la función de coste $J(\theta)$ tratando de maximizarla mediante ascenso de gradientes para obtener así la máxima recompensa posible.

$$J(\theta) = E_p \cdot [\sum_{a_{0:T}} \cdot (\sum_{t=0}^{T} \quad R(x_t, a_t, x_{t+1})) \cdot \pi_\theta(a_{0:T} \mid x_{0:T})]$$

Aplicando el truco del gradiente del logaritmo de una función tenemos lo siguiente: $\nabla_\theta \, J(\theta) = E_\tau[\sum_{t=0}^{T} \quad (U_t \cdot \nabla_\theta \, log \pi_\theta(a_t \mid x_t)]$
Además, podremos hacer una aproximación usando Montecarlo para N episodios:

$$\nabla_\theta \, J(\theta) \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \quad \sum_{t=0}^{T} \quad U_t^{(i)} \cdot \nabla_\theta \, log \pi_\theta(a_t^{(i)} \mid x_t^{(i)})$$

Lo importante de esta aproximación es que con $N = 1$ tenemos una buena aproximación, por lo tanto, con una muestra será suficiente.

$$\widehat{\nabla_\theta \, J(\theta)} \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \quad \nabla_\theta \, J_i(\theta)$$

Los parámetros se actualizarán de la siguiente manera: $\theta \approx \theta - \alpha \cdot \nabla_\theta \, J_i(\theta)$, teniendo que evaluar un episodio diferente cada vez. Esta forma de operar se llama Stochastic Gradient Descent y se muestra a continuación de manera gráfica (Figura 4).

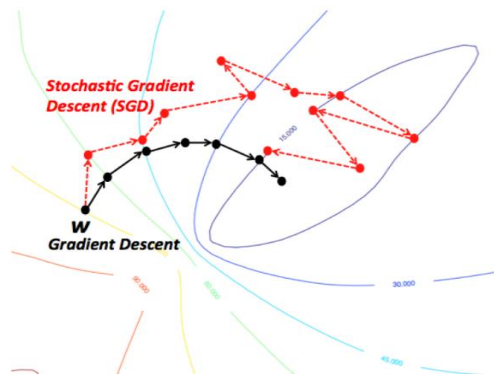


Figura 4. Stochastic Gradient Descent (SGD)

El algoritmo Reinforce [12] actualiza los parámetros mediante Stochastic Gradient Descent y se emplea como base para explicar el algoritmo Actor Critic online.

1) Se recoge una muestra $\{\tau^i\}$ de la política $\pi_\theta(a_t, s_t)$ y se estima las recompensas que se van a recoger siguiendo esa trayectoria.
2) Se actualiza la política con el Stochastic Gradient Descent de la siguiente manera:$\nabla_\theta J(\theta) \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log\pi_\theta(a_t^{(i)} \mid x_t^{(i)}) \cdot (\sum_{t'=1}^{T} r(s_{i,t'}, a_{i,t'}))$
3) $\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J_i(\theta)$

2.3.2. ACTOR: BÚSQUEDA DE POLÍTICA CON FUNCIONES VALOR

Reinforce computa solo una trayectoria dentro de todas las posibles, presenta una alta varianza y necesita realizar episodios completos para actualizar la política. Para solucionar todo esto empezamos a describir la parte del Critic haciendo uso de la función $\hat{Q}_{i,t} \approx \sum_{t=t'}^{T} E_{\pi_\theta}[r(s'_t, a_t')[s_t, a_t]$ para poder tener la recompensa esperada de todas las posibles trayectorias. Se emplea como referencia la media de los valores $Q(s_{i,t}, a_{i,t})$ que es igual a $V(s_{i,t})$ para lograr bajar la varianza y se añade el truco de “bootstrapping”: $\sum_{t'=t}^{T} E_{\pi_\theta}[r(s_{t'}, a_{t'}) \mid s_{i,t}] = r(s_{i,t}, a_{i,t}) + \sum_{t'=t+1}^{T} E_{\pi_\theta}[r(s_{t'}, a_{t'}) \mid s_{i,t+1}]$ para no tener que computar los episodios por completo. Definiremos la función Advantage como:d $\mathrm{A} = Q - S$
Juntando todo lo anterior tenemos la función objetivo que queremos computar:
$\nabla_\theta J(\theta) \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log\pi_\theta(a_t^{(i)} \mid x_t^{(i)}) \cdot A^\pi(s_{i,t}, a_{i,t})$.
El algoritmo Actor-Critic en línea será de la siguiente forma:

1) Se toma una acción $a \sim \pi_\theta(a \mid s)$ obteniéndose un estado, una acción, el siguiente estado y una recompensa. (s, a, s', r)
2) Se actualiza la función valor $\hat{V}_\varphi^\pi(s)$ usando el valor objetivo: $r(s,a) + \gamma \cdot \hat{V}_\varphi^\pi(s')$
3) Se evalúa el Advantage $\hat{A}^\pi(s,a) = r(s,a) + \gamma \cdot \hat{V}_\varphi^\pi(s') - \hat{V}_\varphi^\pi(s)$
4) $\nabla_\theta J(\theta) \approx \nabla_\theta \log\pi_\theta(a \mid s) \cdot \hat{A}^\pi(s,a)$
5) Se actualizan los valores del Actor $\theta \leftarrow \theta + \alpha \cdot \nabla_\theta J(\theta)$

Por lo tanto, el Critic se encargará de aprender los valores φ correspondientes a la función valor $A^{\pi}(s_{i,t}, a_{i,t})$ y El Actor se encargará de los valores θ de la política $log\pi_{\theta}(a_t^{(i)} \mid x_t^{(i)})$.
Aunque los algoritmos Actor-Critic son una buena solución ya que tienen poca varianza debido al Critic, es verdad que introducen algo de sesgo debido a que es estimado secuencialmente.

2.3.3. ESTIMANDO Q CON REDES NEURONALES

Otra manera de obtener la política óptima es a partir de maximizar la función de valor Q.
El aprendizaje de Q con redes neuronales presenta un problema [13]. Si actuamos de manera interactiva las transiciones están correlacionadas, lo cual no es algo positivo.
Para solucionar la correlación de las transiciones se puede hacer uso de “parallel workers" (varios sistemas probando, aprendiendo y compartiendo información en paralelo), esto es lo que se emplea en el algoritmo A2C para que así se obtengan conjuntos de datos totalmente independientes. A veces esto no se puede implementar, por lo que necesitaremos los Replay Buffers D que es una memoria con unos datos previamente recogidos que se emplearán para actualizar la función objetivo. Se recogerán un mini conjunto de datos B aleatorio del Buffer D para tener una serie de datos independientes ya que el Gradiente Estocástico solo funciona con datos independientes y por eso tiene que se B y no D.
El algoritmo de Q learning con Replay Buffers quedará de la siguiente manera:

1) Recogemos una colección de datos $D = \{(s_i, a_i, s_i', r_i)\}$ empleando una política y lo almacenamos en el Buffer.
2) Recogemos un mini-batch de datos B $\{(s_i, a_i, s_i', r_i)\}$ de manera aleatoria del Buffer (D).
3) Actualizamos los features de la red neuronal con la función objetivo, pero ahora emplearemos este truco del “mini-batch” para el aprendizaje por refuerzo: $\varphi \leftarrow \varphi - \alpha \cdot \sum_i \frac{dQ_{\varphi}}{d\varphi} \cdot (s_i, a_i) \cdot (Q_{\varphi}(s_i, a_i) - r(s_i, a_i) + \gamma \cdot max_{a_i'} Q_{\varphi}(s_i', a_i'))$

Se utiliza también el doble Q Learning, es decir, se utilizan dos redes diferentes para estimar Q. Una que se actualiza continuamente y otra que se actualiza con poca frecuencia y se usa para calcular $maxQ$ tratando de evitar la sobreestimación de la función Q al emplear expectaciones de la función de valor Q dadas por las redes neuronales y no funciones Q verdaderas. Esto puede verse matemáticamente en la siguiente expresión: $E[max(X_1, X_2)] \geq max(E[X_1], E[X_2]$. Estamos empleando $E[max(X_1, X_2)]$ como aproximación por lo que a veces sobreestimaremos los valores Q , para ello se utilizan dos redes Q (double Q-Learning).
El DDPG, TD3 y el SAC realmente son algoritmos que se estudian como Actor-Critic, pero se pueden considerar como algoritmos Q-Learning con un agente que maximizar ya que su código interno tiene dos partes. El Critic consta de una red neuronal que se encarga de aprender la función Q y el Actor consta de otra red neuronal diferente que se encarga de maximizar esta función Q.

2.4. ALGORITMOS EMPLEADOS

En el trabajo se emplean los algoritmos A2C, PPO, TD3 y SAC, pero se va a realizar la explicación especialmente del SAC ya que éste es el que se ha empleado tras la obtención de los resultados de las simulaciones. El PPO es el algoritmo más popular de Policy Gradient que puede ser usado para entornos con estados de actuación discretos o continuos y además la implementación de SpinningUP permite "parallelization". Es "On-Policy", es decir todas sus acciones provienen de la política que se quiere obtener. El algoritmo A2C también es de la familia de los "On Policy" y trata de optimizar la política sobre la base de Policy Gradient. Es un Actor Critic pero que trabaja con "parallel workers" para evitar hacer uso de los Replay Buffers. Tanto el PPO como el A2C siguen una política determinista por lo que necesitan la inclusión de ruido durante el entrenamiento para poder tomar acciones aleatorias para mejorar el aprendizaje del agente.

2.4.1. SAC

La característica principal del Soft Actor Critic es la incorporación de la regularización de entropía, la cual es una medida de aleatoriedad de una variable.
Hay que destacar que el algoritmo SAC:

- Tiene una estructura Actor Critic con tres redes (π, Q_1, Q_2), un Actor y dos Críticos separadas para calcular la función de valor y la política
- Es un algoritmo "off-Policy": Este evalúa y mejora la política objetivo mediante la utilización de otra política diferente para la selección de acciones lo cual facilita explorar acciones nunca vistas.
- La versión utilizada en este trabajo es una versión apta para espacios continuos donde es muy difícil calcular el $maxQ^*$. La función óptima $Q^*(s,a)$ es diferenciable con respecto a las acciones, por lo que nos basaremos en el gradiente para una política $\mu(s)$. Aproximaremos $maxQ(s,a) \approx Q(s,\mu(s))$ siendo $\mu(s)$ una red neuronal con parámetros φ.

En el aprendizaje por refuerzo con regularización de entropía, el agente obtiene una recompensa extra en cada "time step" de manera proporcional a la entropía de la política. Esto se hace porque la entropía es una medida de aleatoriedad con la que se pretende tratar de aumentar la exploración del agente, consiguiendo así un mejor entrenamiento. Por lo que ahora el problema de aprendizaje por refuerzo cambia un poco y la política óptima que se quiere calcular será de la siguiente manera:

$$\pi^* = argmax\, E[\sum_{t=0}^{\infty} \gamma^t \cdot (R(s_t, a_t, s_{t+1}) + \alpha \cdot H(\pi(s_t)))]$$

α es el coeficiente de compensación.
La definición de las funciones $V^\pi(s)$ y $Q^\pi(s,a)$ será ligeramente diferente:

$$V^\pi(s) = E[Q^\pi(s,a)] + \alpha \cdot H(\pi(s_t))$$

$$\Rightarrow Q^\pi(s,a) = E[R(s,a,s') + \gamma \cdot V^\pi(s')]$$

Si tenemos una variable cualquiera x con una función de densidad P, entonces su entropía H será: H(P)=E [-log P(x)].
A partir de ahora la notación empleada en esta sección cambiará ligeramente, a la próxima acción se le llamará $\tilde{a}'$ en vez de a' . Esto se hace para remarcar que las acciones siguientes tienen que provenir de la política, en cambio r y s' tienen que venir del Replay Buffer porque se trata de un algoritmo off-Policy y, por lo tanto, la ecuación de Bellman de la función Q será la siguiente: $Q^{\pi}(s,a) = E[R(s,a,s') + \gamma \cdot (Q^{\pi}(s',a') - \alpha \cdot log\pi(a' \mid s'))]$
$\Rightarrow Q^{\pi}(s,a) \approx r + \gamma \cdot (Q^{\pi}(s',\tilde{a}') - \alpha \cdot log\pi(\tilde{a}' \mid s'))]$
Tenemos que definir un error medio cuadrado de Bellman $L(\varphi_i, D)$ el cual nos indicará si $Q(s,a)$ se aproxima bien a la ecuación de Bellman. Esta será la función que se trata de minimizar en el SAC: $L(\varphi_i, D) = E[(Q_{\varphi_i}(s,a) - y(r,s',d))^2]$
Siendo $y(r,s',d) = r + \gamma \cdot (1-d) \cdot (min\ _{j=1,2} Q_{\varphi j_{targ}}(s',\tilde{a}') - \alpha \cdot log\pi_{\varphi}(\tilde{a}' \mid s'))$
$(1-d)$ modela cuando el estado es terminal ($d=1$).
Se emplea el truco del doble-Q escogiendo el mínimo valor Q para los dos aproximadores de Q. $Q\ (min\ _{j=1,2} Q_{\varphi j_{targ}}(s',\tilde{a}'))$. µ y $Q_{\varphi_{target}}(s',\tilde{a}')$ son dos redes neuronales objetivo para computar aproximadamente $maxQ^*(s',a')$. Para aprender la política se maximizan las recompensas futuras esperadas y la entropía futura esperada. $V^{\pi}(s) = E[Q^{\pi}(s,a) + \alpha \cdot H(\pi(\cdot \mid s))] = E[Q^{\pi}(s,a) - \alpha \cdot log\pi(a,s)]$.
El truco de la parametrización, permite reescribir la expectación:

$$E[Q^{\pi_\theta}(s,a) - \alpha \cdot log\pi_\theta(s,a)] = E[Q^{\pi_\theta}(s, \underline{a}_\theta(s,\xi)) - \alpha \cdot log\pi_\theta(\underline{a}_\theta(s,\xi) \mid s)]$$

Por lo tanto, para obtener la política el paso final es emplear el truco de doble Q, mencionado también antes. La política se optimizará de acuerdo con:

$$\max\ E[(\min\ _{j=1,2} Q_{\varphi j}(s, \underline{a}_\theta(s,\xi)) - \alpha \cdot log\pi_{\varphi}(\underline{a}_\theta(s,\xi) \mid s')]$$

3. ENTRENAMIENTOS Y RESULTADOS

Se decidió que las librerías empleadas fuesen las siguientes:

- `stable-baselines3[extra]`: Instala las funciones del Github de Stable Baselines 3.
- `pybullet`: Necesaria para hacer uso de los entornos de Pybullet.
- `ffmpeg`, `freeglut3-dev`, `xvfb`: Se usa para la visualización de los resultados desde Colab
- `tensorboard`: Indispensable para poder emplear Tensorboard y almacenar todos los datos del entrenamiento.

Es necesario el uso de funciones ya creadas ya que este trabajo es de análisis y no de implementación de código propio. Se necesita importar funciones tanto provenientes de Stable Baselines 3 como de otros directorios.

3.1. ENTORNOS

Se ha estudiado el comportamiento de los algoritmos Actor-Critic (A2C, TD3 y SAC) con uno de referencia (PPO) y para ello se han usado diferentes entornos con diferentes grados de complejidad. La elaboración de los entornos se ha realizado empleando las funciones de Gym, DummyVecEnv y Monitor y os que es un módulo de Python para el sistema operativo, como se puede ver en el código de Google Colab. Estas sirven para crear el entorno, vectorizarlo y monitorizar los resultados respectivamente. Los resultados también fueron almacenados en un archivo para Tensorboard, pero para esto no fue necesario emplear ninguna función especial ya que los propios algoritmos de Stable Baselines 3 llevan incorporados la posibilidad de crear un archivo si lo configuras. Se realizaron pruebas con otros entornos siguiendo los tutoriales de Stable Baselines 3 antes de la realización del análisis de los algoritmos. Estos entornos empleados son de OpenAI Gym y fueron impuestos por los tutoriales realizados ("Lunar Lander", "Parking" y "Pendulum") y por eso no se hace una especial mención de ellos.

En el análisis de los tres algoritmos se han empleado los entornos: 'CartPole-v0', 'CartPole-v1', 'CartPoleContinuousBulletEnv-v0', 'HalfcheetahBulletEnv', 'HopperBulletEnv' y 'AntBulletEnv-v0'. El entorno más simple utilizado es el 'CartPole' de OpenAI (Figura 5) el cual se empleó durante la familiarización con las herramientas y para ver por primera vez cómo los diferentes hiperparámetros y estructuras de cada algoritmo afectaban a la velocidad y a la calidad del aprendizaje. Es una versión de CartPole que simplemente simula "un palo" y "un carrito" en la que se tiene en cuenta la inclinación respecto al eje vertical, pero no la gravedad ni el rozamiento que puede existir en un ambiente real. El sistema se controla aplicando una fuerza de +1 o -1 al carro. El péndulo comienza en posición vertical y el objetivo es evitar que se caiga. Se proporciona una recompensa de +1 por cada paso de tiempo que el poste permanezca en posición vertical. El episodio termina cuando el poste se aleja más de 15 grados de la vertical, o el carro se desplaza más de 2,4 unidades del centro.

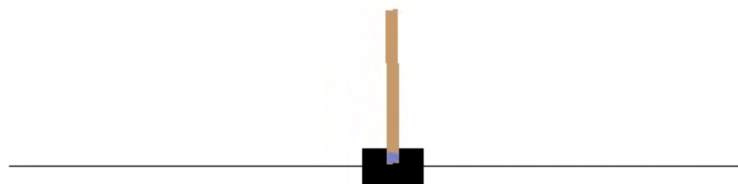


Figura 5. Entorno CartPole de OpenAI

Hay que destacar que se han empleado tanto 'CartPole-v0' como 'CartPole-v1' de Open AI y cada uno dispone de una configuración en cuanto a las acciones, a la recompensa y a la duración del episodio. Para el 'CartPole-v0' el máximo número de pasos por episodio será 200 y para el 'CartPole-v1' será de 500.

Los algoritmos A2C y PPO funcionan con entornos cuyos estados de actuación son tanto discretos como continuos, en cambio la implementación disponible del TD3 y el SAC funcionan con estados de actuación continuos, por lo tanto, se ha empleado la versión CartPole continua de Pybullet 'CartPoleContinuousBulletEnv-v0' (Figura 6) que tiene una configuración que lo hace un entorno más complejo. La recompensa máxima por episodio es de 200. Se trata de una versión que emplea un motor físico 3D por lo tanto se tienen en cuenta muchos parámetros como la gravedad, el rozamiento, el movimiento 3D o la velocidad de movimiento de manera más compleja que el entorno de OpenAI.

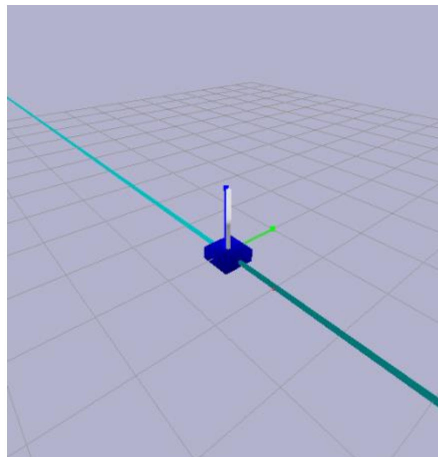


Figura 6. Entorno CartPole de Pybullet

Por último, se escogieron los entornos 'HalfcheetahBulletEnv-v0', 'HopperBulletEnv-v0' y 'AntBulletEnv-v0' de Pybullet que son realmente más complejos y es donde se estudió el comportamiento del Soft Actor Critic (SAC) siendo el objetivo de estas tareas que avancen lo máximo posible. El máximo número de pasos por episodio para los tres es de 1.000 pasos en cambio el umbral de recompensa es de 2.500 para el 'Hopper' y para el 'Ant' y 3.000 para el 'Halfcheetah'.

El espacio de estado del Hopper es continuo y tiene una dimensión de 15, es decir, tiene 4 elementos con 3 grados de libertad (el resto de dimensiones son para otros elementos). El espacio de acción es de 3 dimensiones continuas porque posee 3 juntas (Figura 7). El espacio de estado del HalfCheetah es continuo y de dimensión 26, es decir, tiene 7 elementos con 3 grados de libertad (el resto de dimensiones son para otros elementos). El espacio de acción es de 6 dimensiones continuas ya que tiene 6 juntas (Figura 8). Las recompensas se dan en cada paso en función de sus movimientos relativos con respecto al suelo, al coste de control y a su posición. Nótese que todas estas recompensas mencionadas anteriormente sólo se utilizan para evaluar el rendimiento de los agentes y generar demostraciones, pero nunca para el aprendizaje. Como hemos visto, la simulación tiene en cuenta múltiples elementos y su forma de unión mediante juntas que permiten simular el entorno de una manera muy real.

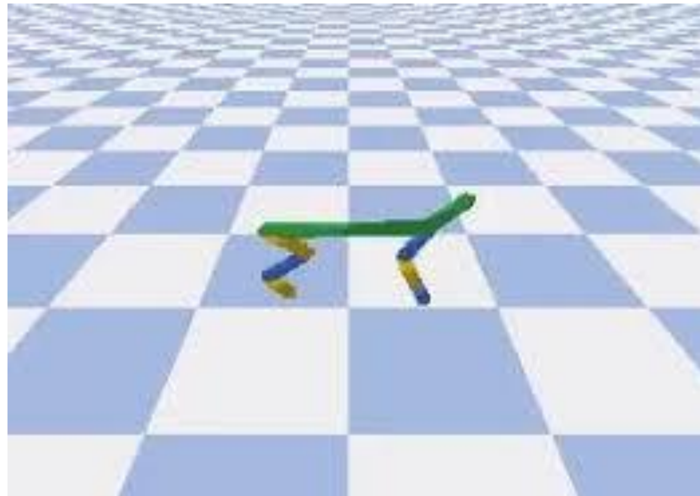


Figura 7. Entorno de Halfcheetah de Pybullet

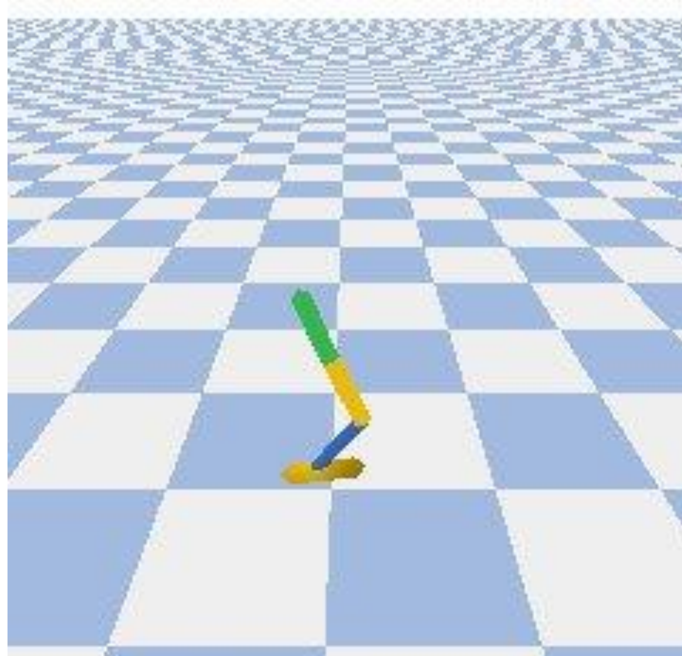


Figura 8. Entorno de Hopper de Pybullet

Este último entorno 'AntBulletEnv-v0' es más difícil de entrenar ya que se trata de una araña con movimiento 3D en cambio en los dos anteriores se asumía un movimiento 2D y por lo tanto no podían caerse hacia los lados. Al asumir 3D y la posible rotación tenemos un espacio de estado de dimensión 51 ya que tiene 9 elementos con 3 grados de libertad. El resto de dimensiones son para las juntas, para el centro de masas y otros elementos (Figura 9).

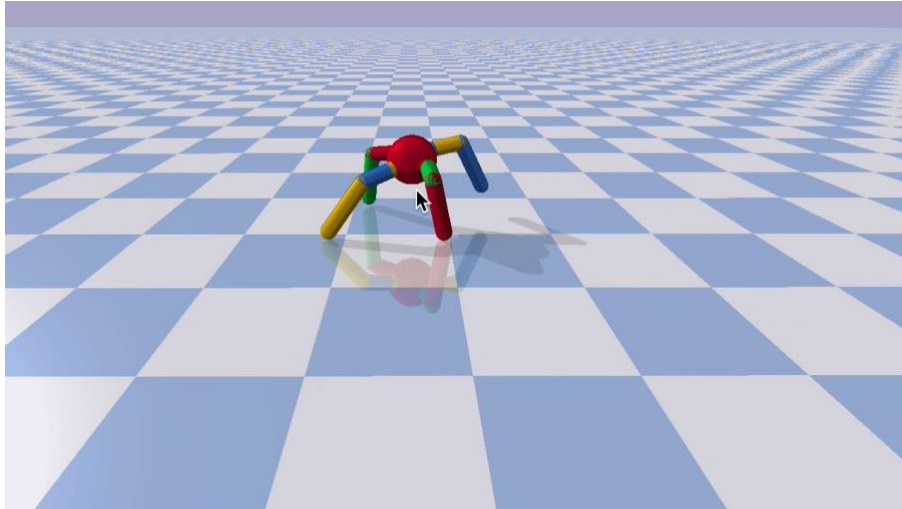


Figura 9. Entorno de Ant de Pybullet

Hay que destacar que se han empleado entornos en un simulador robótico por comodidad. El CartPole sería más fácil de implementar en el mundo real, en cambio el Halfcheetah, el Hopper y el Ant exigen unos recursos y unas instalaciones más sofisticadas para su funcionamiento.

3.2. CARTPOLE DE OPEN AI GYM

Las primeras simulaciones realizadas fueron en el entorno CartPole de Open AI ya que éste era el menos complejo y así los entrenamientos eran relativamente rápidos, ideal para empezar a familiarizarse con las herramientas empleadas.
Cabría destacar que durante estos primeros entrenamientos se ha visto que la recompensa obtenida con un entrenamiento de 20.000 pasos, aunque era aceptable no era todo lo buena que debía ser según nos indicaba la función evaluate_policy. Esta es una función propia de Stable Baselines que devuelve la recompensa media y la desviación típica tras realizar una evaluación de cinco episodios del algoritmo entrenado.
Esta discordancia entre valores ocurría porque, como se explica en la documentación, durante el entrenamiento se activa ruido para que el agente pueda explorar acciones nuevas y esto hace que el comportamiento no sea óptimo por lo que las recompensas no eran todo lo buenas que se esperaba. Para solucionar este problema se decidió realizar en un entorno idéntico al de entrenamiento una evaluación simultánea y así poder comparar la recompensa obtenida durante el entrenamiento y durante la evaluación. Los algoritmos empleados tienen una función interna que, como se indica en la documentación, por defecto está desactivada, pero se puede activar para crear el entorno de evaluación de manera sencilla. Esto se ha empleado para todos los algoritmos en todos los entornos.

Realmente estas primeras simulaciones no tienen mayor trascendencia más allá del aprendizaje y el dominio de las herramientas necesarias. Por lo tanto, se muestra la recompensa obtenida durante el entrenamiento del algoritmo PPO tanto en la versión discreta (CartPole-v1) como en la continua (CartPole-v0), pero no se hace especial hincapié en otros resultados.

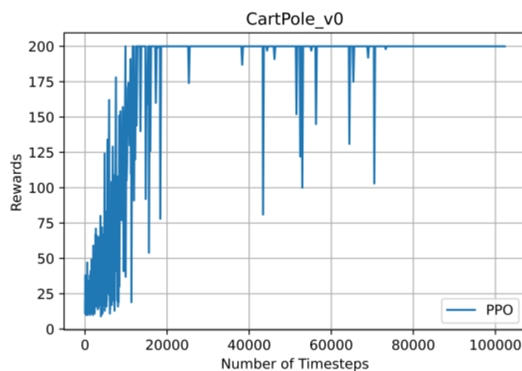


Figura 10. Evaluación de CartPole-v0

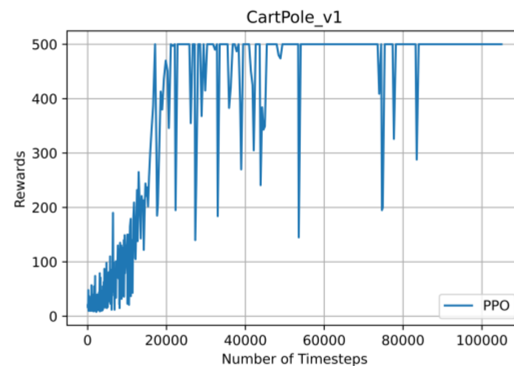


Figura 11. Evaluación de CartPole-v1

La recompensa obtenida como era de esperar fue de 200 (Figura 10) y 500 (Figura 11) que es el valor máximo posible para los respectivos entornos. Se decidió realizar 100.000 timesteps porque resultaba ser un entrenamiento rápido y se quería ver los resultados de entrenar para bastantes “timesteps” pero 20.000 son suficientes.
Ya que se estaba aprendiendo el funcionamiento del código y de los algoritmos y el entrenamiento era rápido se realizaron bastantes más simulaciones para asegurar que era un aprendizaje robusto y que no llegaba a esa solución por casualidad.
Una vez hecha la familiarización con las herramientas y los entrenamientos se procedió durante el resto del trabajo a evaluar los algoritmos para ver cuál es el mejor de acuerdo a los resultados obtenidos, realizándose las ejecuciones en el entorno de Pybullet. Más concretamente se trata del entorno de CartPole de Pybullet que comienza desde arriba.

3.3. CARTPOLE DE PYBULLET

Se utilizó MLP Policy que es una “multi-layer percepton” con 2 capas ocultas de 64 neuronas que nos permitirá aproximar directamente una función de política en lugar de una función de valor. Es decir, en lugar de entrenar una red neuronal para que genere los valores de las acciones o estados, entrena una red neuronal que indica cual puede ser la siguiente acción a realizar. Este entorno es más complejo que el de Open AI ya que está construido con un motor físico en el que se tienen en cuenta una serie de parámetros como la fuerza de gravedad, el rozamiento y el movimiento en 3D. Esto hace que sea un entorno mucho más realista y por lo tanto más difícil de entrenar.

3.3.1. A2C

Se empezó a trabajar empleando 40.000 "timesteps" para analizar el comportamiento del agente y evaluar el algoritmo. Las gráficas para ver los resultados corresponden a los entornos de evaluación del entrenamiento en el que no hay ruido activado.

La Figura 12 nos muestra claramente lo que ya nos esperábamos: El algoritmo A2C al ser el que menos parámetros tiene y el más sencillo, cuando los entornos se complican le cuesta mucho más trabajar correctamente.
No es muy probable que con la variación de hiperparámetros acabase mejorando su comportamiento ya que en principio los parámetros están ajustados de serie para funcionar lo mejor posible. Se puede ver que en ningún momento se llega a la recompensa de 200 que es máximo alcanzable.

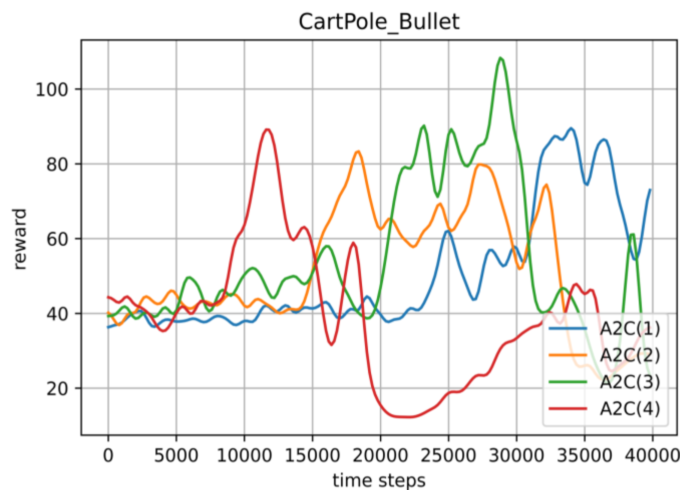


Figura 12. Cuatro evaluaciones del algoritmo A2C

Para comprobar si al entrenar durante más pasos se obtenían mejores resultados se entrenó el modelo durante 100.000 "timesteps" (Figura 13).
Se ve con claridad que no siempre entrenar durante más tiempo es la mejor solución para obtener mayores recompensas, ya que por mucho que entrenemos este algoritmo para este entorno no mejorará la recompensa obtenida.
Esto es algo que fue de gran utilidad para los entrenamientos realizados posteriormente que al ser muy largos tardan mucho tiempo en entrenar. Es importante señalar que en muchas ocasiones es mejor focalizarse en otros parámetros.

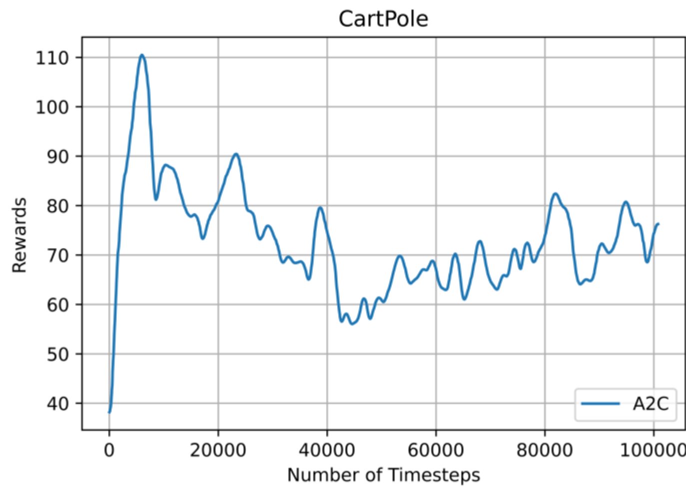


Figura 13. Evaluación del entrenamiento de 100.000 time steps

3.3.2. PPO

Este algoritmo aprende bastante rápido, igual que el explicado anteriormente, por lo cual se realizaron simulaciones con más pasos para ver cómo aprendía y para ver si el entrenamiento era robusto.
Cabe destacar que el aprendizaje es más estable que el A2C y aunque al principio le cuesta cierto tiempo conseguir buenas recompensas a los 40.000 pasos llegamos a obtener la recompensa esperada para las 4 evaluaciones mostradas. (Figura 14)
A priori, si el entorno que fuéramos a entrenar en el futuro fuese de una complejidad parecida a la del CartPole de Pybullet quizá el PPO sería una buena solución ya que, aunque le cuesta bastantes “time steps" llegar a una buena recompensa el entrenamiento es muy rápido.

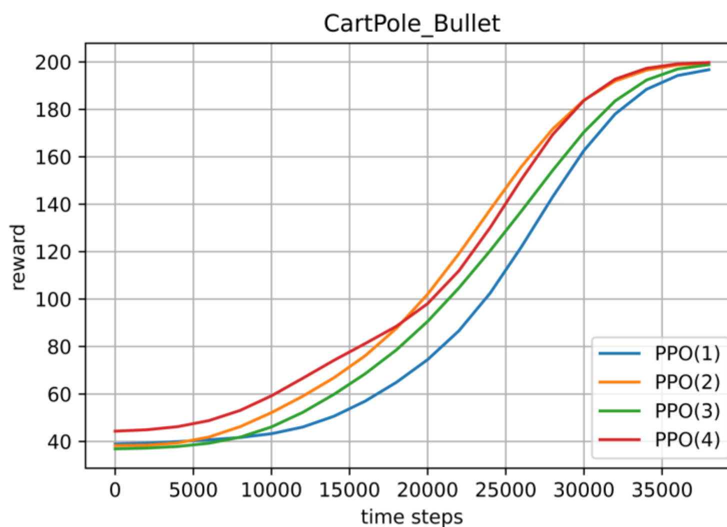


Figura 14. Cuatro evaluaciones del Algoritmo PPO

3.3.3. TD3

Este algoritmo es más complejo y emplea más trucos (doble Q Learning, actualización de la política con retraso y suavizado de la política objetivo) para poder así mejorar el comportamiento del agente. Se podría decir a priori que es mejor algoritmo que los otros dos ya mencionados, aunque los resultados en la Figura 15 no nos muestran lo mismo. Para los entrenamientos (3) y (4) la recompensa obtenida es bastante buena a los 35.000 “time steps”, pero para los entrenamientos (1) y (2) la recompensa nunca llegó a ser buena.
Aunque sobre el papel este algoritmo debería trabajar mejor que los anteriores no ha ocurrido esto. Además, al TD3 le cuesta bastante más tiempo entrenar que al A2C y al PPO, por lo tanto, no será el algoritmo escogido para las futuras pruebas.

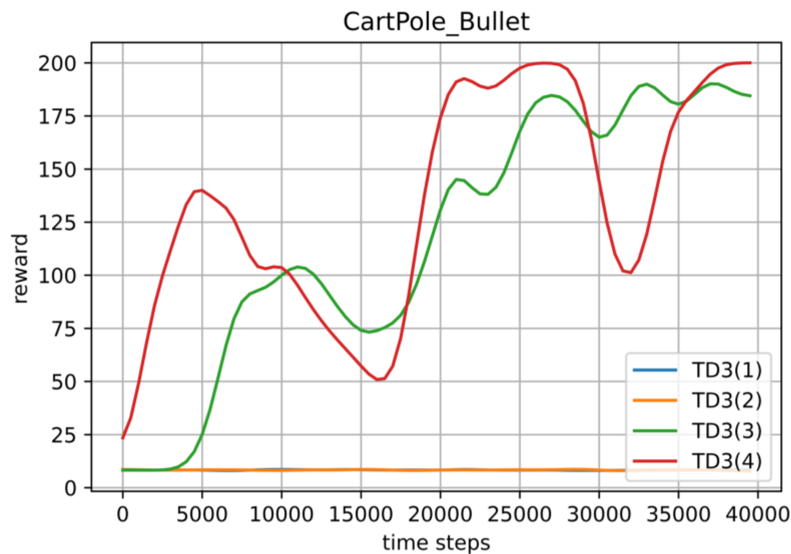


Figura 15. Cuatro evaluaciones del algoritmo TD3

3.3.4. SAC

Teniendo en cuenta todas las técnicas que emplea el SAC para que el algoritmo no falle y aprenda de la forma más robusta posible, este debería de ser en teoría el algoritmo más adecuado para entrenar tareas más complejas. Los resultados como podemos ver en la Figura 16 nos corroboran la suposición.
Por lo general el SAC a los pocos “time steps” (15.000) obtiene buenos resultados y será idóneo para tareas más complejas, aunque le cueste más tiempo el entrenamiento. Este tiempo invertido merece la pena por los resultados obtenidos.

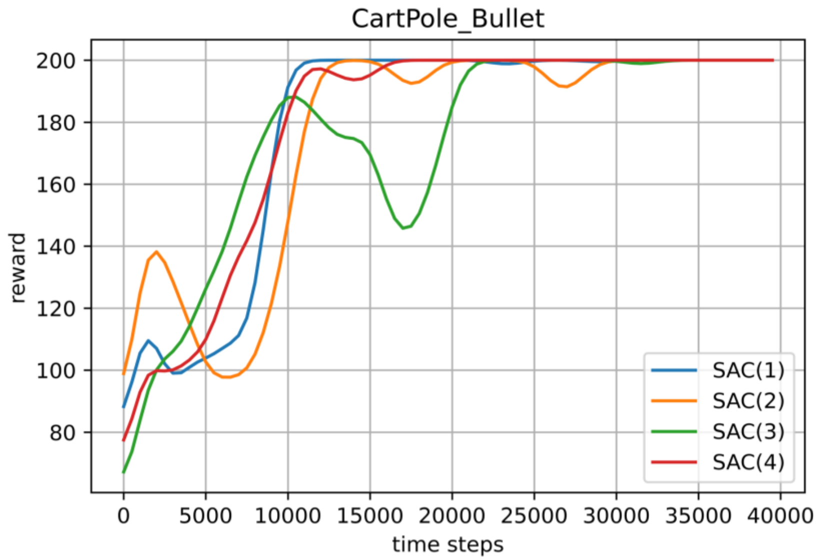


Figura 16. Cuatro evaluaciones del algoritmo SAC

En la Figura 16 se representan los entrenamientos y las evaluaciones de los 4 algoritmos en los mismos ejes. Se puede intuir qué algoritmo será el mejor para realizar las simulaciones complejas. Se hace una media de los 4 entrenamientos mostrados anteriormente para cada algoritmo y se obtiene un entrenamiento medio, lo cual muestra resultados más fiables. Si por casualidad en un entrenamiento se han obtenido recompensas buenas o malas por azar, éstas se compensarán con el resto de entrenamientos.

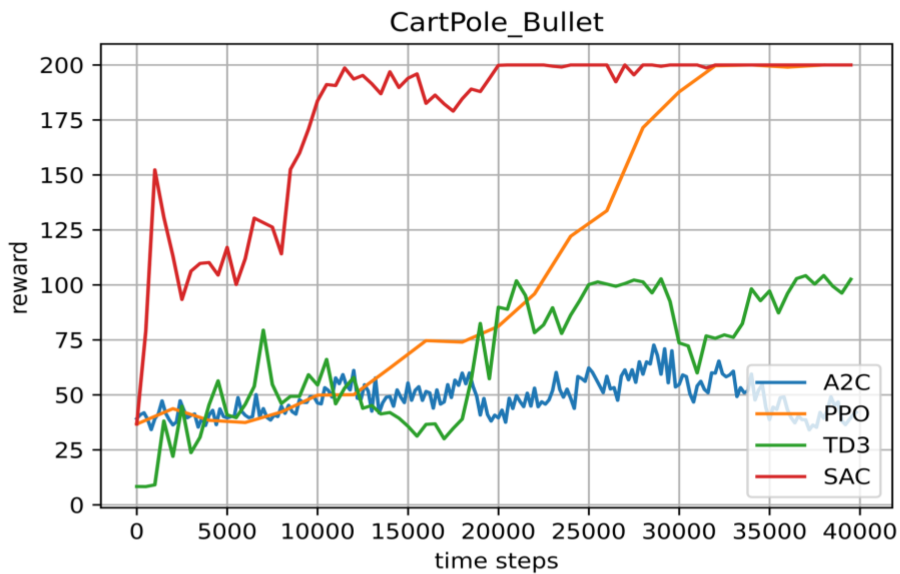


Figura 17. Evaluación de los cuatro algoritmos

Ahora se va a representar en la Figura 17 la gráfica de los entrenamientos medios, pero mostrando la desviación típica para ver la dispersión de las recompensas esperadas.
Se ha empleado para ello la función de “error bar” de Matplotlib mencionada anteriormente.

Si pudiesen existir dudas entre qué algoritmos emplear si el PPO o el SAC ya que ambos muestran buenos resultados en la evaluación de los entrenamientos, podemos comprobar gracias a la Figura 18 que el algoritmo SAC funciona muy bien. En cambio, el PPO no ha dado tan buen resultado, aunque a priori por las gráficas vistas anteriormente nos indicaban que si. Para un 35.000 time steps el SAC tiene menos desviación típica que el PPO por lo tanto en general recibiremos siempre mejor recompensa. La desviación típica del A2C es muy pequeña, pero en este caso no nos interesa ya que, aunque tendrá pocas variaciones de recompensa, estarán todas entorno a 50 lo cual no es un resultado aceptable.

Finalmente se ha decidido, tanto teórica, como empíricamente que el Soft-Actor-Critic (SAC) será el algoritmo empleado para evaluar los entornos de Pybullet, ‘Halfcheetah’, ‘Hopper’ y ‘Ant’.

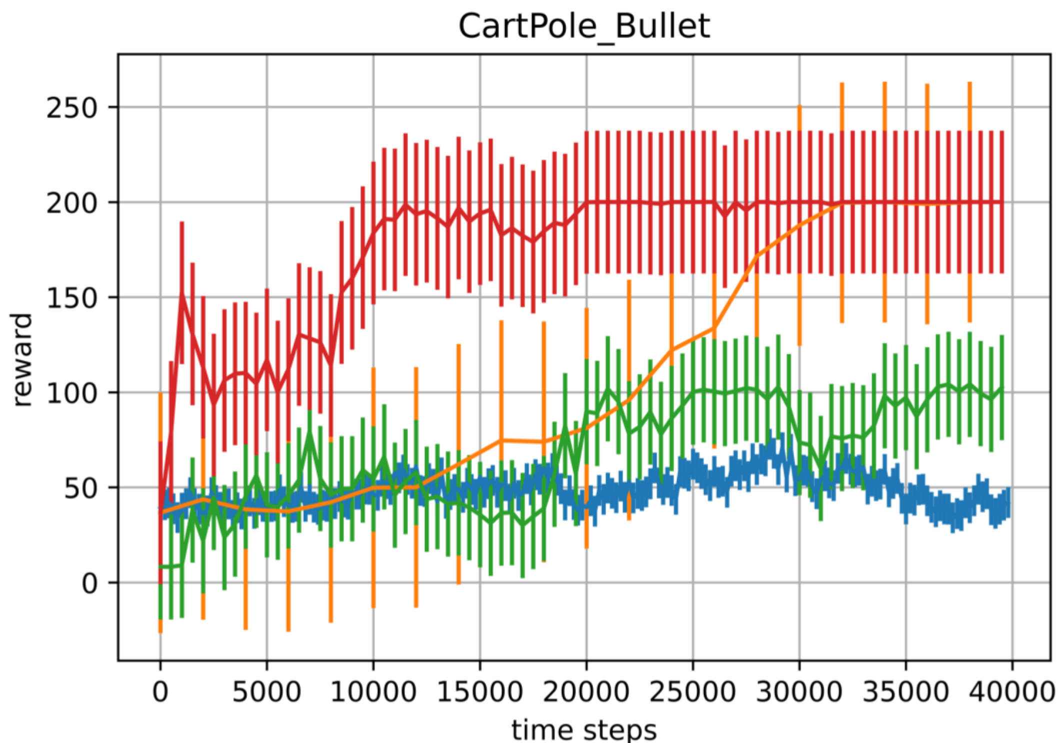


Figura 18. Evaluación de los cuatro algoritmos con ruido

3.4. HALFCHEETAH, HOPPER Y ANT

Estos son entornos continuos de Pybullet más complejos que el CartPole estudiado anteriormente, en cuanto a las observaciones y a las acciones que tiene que tomar el agente. Es necesario simular el movimiento de un mayor número de juntas y elementos. Se trata de una simulación mediante el motor físico de Pybullet de un Guepardo y de un Robot con forma de pierna y la tarea que tienen que aprender es la de correr.

Por lo aprendido durante el estudio de los algoritmos y su comportamiento se decidió emplear el algoritmo SAC para evaluar estos dos entornos. Se hizo uso de un entorno de evaluación paralelo para evaluar el agente entrenado sin el ruido del entrenamiento. Se realizaron primero, unos cuantos entrenamientos para muy pocos “timesteps” a modo de prueba ya que había que asegurarse que se guardaba toda la información que se pretendía. Al ser entrenamientos bastante largos tenía que estar todo correctamente ajustado desde el principio. Una vez se comprobó que el código de entrenamiento de Halfcheetah, Hopper y Ant funcionaba correctamente se procedió con la realización del análisis del SAC para los dos entornos. Se realizaron varios entrenamientos del agente para asegurarnos de que éstos eran robustos y no se llegaba a obtener buenos resultados por casualidad. Se hizo una media de los resultados y se obtuvo un entrenamiento medio para el Halfcheetah y otro entrenamiento medio para el Hopper.

Los datos de estos experimentos se guardaron de manera local y una vez se dispuso de la información que se necesitaba se procedió a realizar las gráficas. Los resultados han sido bastante buenos ya que como sabíamos por la documentación, para 200.000 “time steps” tanto el Halfcheetah como el Hopper deberían obtener una recompensa media de alrededor de 1.500.

Se puede ver que el Halfcheetah (Figura 19) obtiene una recompensa de entre 1.200 y 2.300 y el Hopper (Figura 20) de entre 1.500 y 2.600. Mediante la visualización de los videos de la evaluación de los modelos se comprobó que con la recompensa obtenida para los valores anteriormente citados los agentes avanzaban y no se caían.

Se ha realizado una última evaluación del SAC para un ambiente más complejo que el de Hopper y HalfCheetah ya que éstos asumían el movimiento 2D y por eso no se caían hacia los lados. El ambiente 3D seleccionado ha sido el de ‘AntBulletEnv-v0’. El SAC es un algoritmo que tarda bastante en entrenar al agente por lo que se decidió entrenar para 668.000 “time steps”. Se ve cómo para el mismo numero de ‘time steps’ que habíamos realizado anteriormente (200.000), las recompensas obtenidas son muy malas, lo cual supondrá un mal comportamiento del agente entrenado (Figura 21). Esto es algo que podríamos esperar ya que ahora el entorno es más complejo y hay más variables a tener en cuenta.

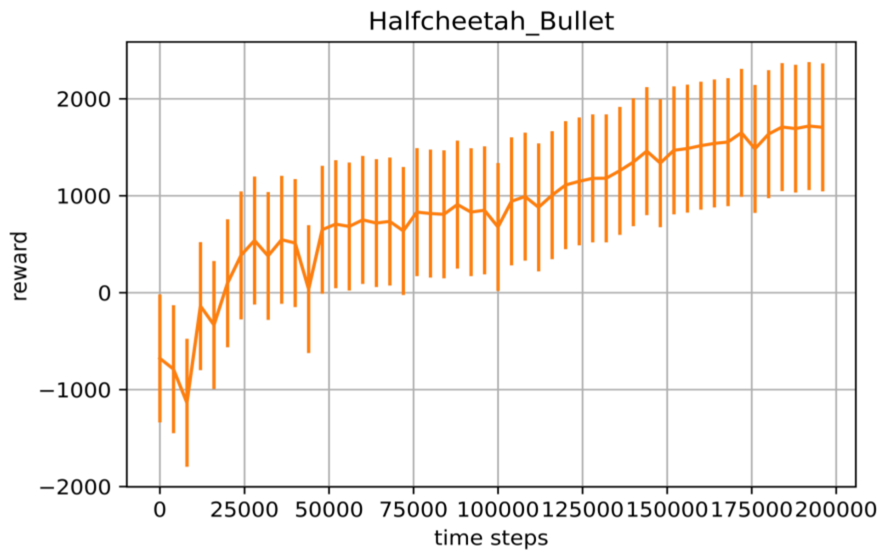


Figura 19. Evaluación del entrenamiento con SAC

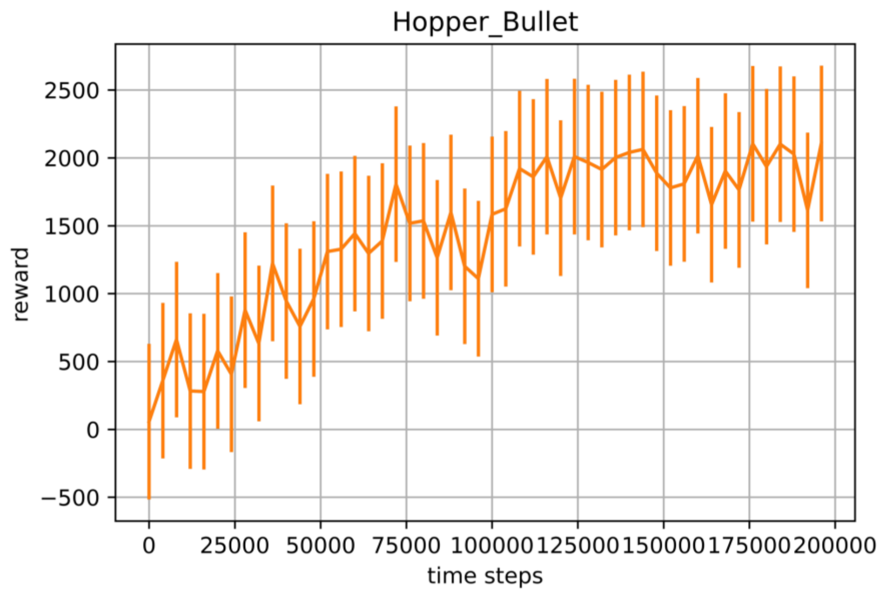


Figura 20. Evaluación del entrenamiento con SAC

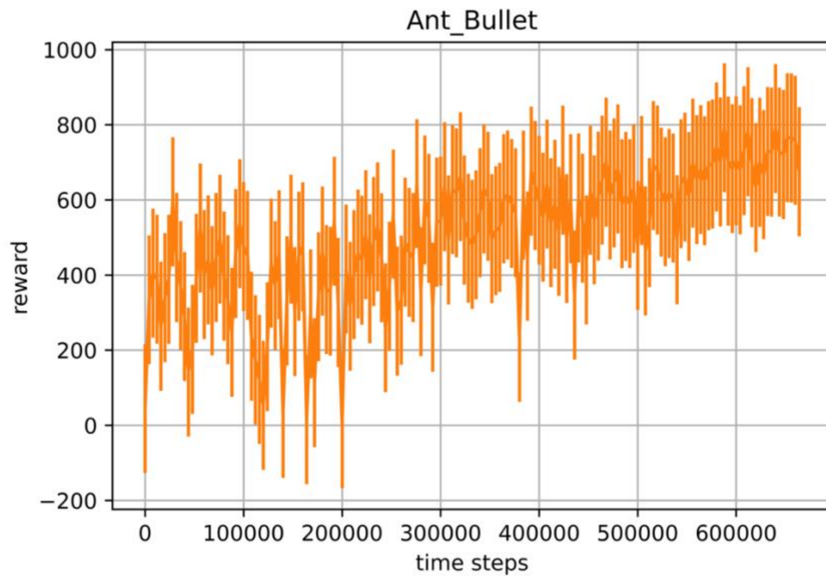


Figura 21. Evaluación del entrenamiento con SAC

Habiendo entrenado para más número de pasos hemos podido visualizar como el agente (araña) 'AntBulletEnv-v0' puede empezar a andar sin caerse (Figura 22).

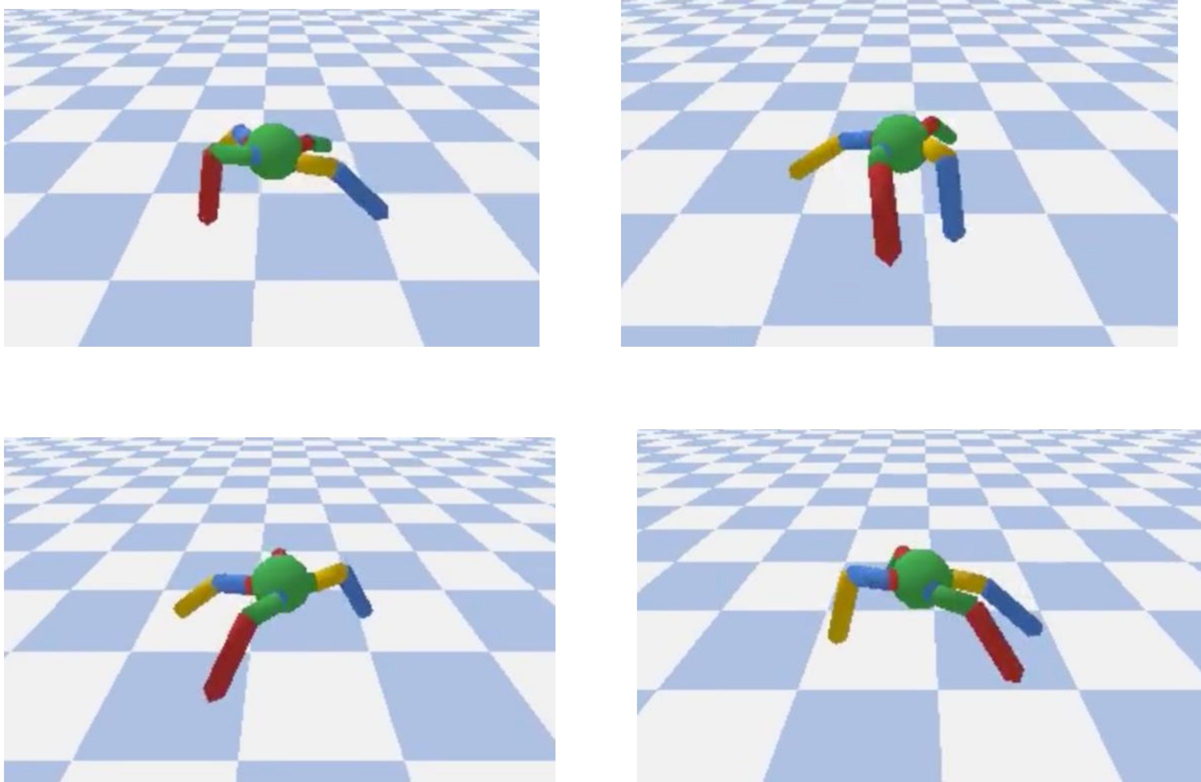


Figura 22. Renderizado de la evaluación de 'AntBulletEnv-v0'

CONCLUSIONES

Se ha llevado a cabo la evaluación de algoritmos de aprendizaje por refuerzo mediante el análisis de una biblioteca de tareas robóticas en un simulador físico 3D.

Se muestra tanto teórica como empíricamente cómo de importante es elegir los algoritmos de acuerdo a las tareas que se quieren desarrollar. Los diferentes algoritmos Actor Critic se implementan de manera diferente siempre siguiendo el concepto de tener dos redes neuronales para actuar y evaluar. Aunque estas variaciones los hacen más complejos y por lo tanto el entrenamiento será más costoso los resultados que arrojan son mucho más fiables y correctos.

En las tareas simples como el CartPole en de OpenAI no es necesario el uso de los algoritmos más complicados ya que el entrenamiento será muy lento y con algoritmos simples como A2C es suficiente para la obtención de buenos resultados. En cambio, en entornos como los de Pybullet que emplean más elementos y juntas para simular el comportamiento del agente, además de tener en cuenta variables como la gravedad, el rozamiento y el movimiento de los elementos y su centro de masas será necesario el uso de algoritmos como el SAC o el TD3.

La utilización de algoritmos complejos (SAC y TD3) que emplean el aprendizaje del doble valor Q, actualizan la política con retraso, realizan un suavizado de la política objetivo y emplean la regularización de entropía nos ha permitido obtener un mejor entrenamiento y comportamiento del agente en entornos.

Bibliografía

[1] R. S. Sutton and A. G. Barto, Reinforcement Learning, The MIT Press, 2018, pp. 2-4.

[2] D. Hassabis, "Deepmind," 2017. [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[3] A. Hill, "Stable Baselines 3," 2020. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/.

[4] E. Coumans and C. , "Bullet Physics SDK," 2021. [Online]. Available: https://github.com/bulletphysics/bullet3.

[5] G. Brockman and Cols, "OpenAI Gym," 2016. [Online]. Available: https://github.com/openai/gym/blob/master.

[6] E. Coumans, "Pybullet Quickstart Guide," [Online]. Available: https://usermanual.wiki/Document/pybullet20quickstart20guide.479068914/html#pf2.

[7] A. Hill, "Stable Baselines," 2018. [Online]. Available: https://stable-baselines.readthedocs.io/en/master/index.html.

[8] "NumPy," 22 6 2021. [Online]. Available: https://numpy.org/doc/stable/reference/.

[9] "MatplotLib," 13 7 2021. [Online]. Available: https://matplotlib.org/stable/users/index.html.

[10] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/.

[11] C. Yoon, "Understanding Actor Critic Methods and A2C," 2 2019. [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f.

[12] D. Silver, "Lectures on Reinforcement Learning," 2015. [Online]. Available: https://www.davidsilver.uk/teaching/.

[13] R. Martinez Cantin, "Model free methods," in *Applications of Deep Learning* , 2021.

A. Anexo A:

a. Procesos de decisión de Markov

Se explican con carácter general para luego hacer algunas suposiciones y dar así lugar a los problemas reales del aprendizaje por refuerzo. Esto es necesario ya que estos procesos son descripciones formales del entorno de aprendizaje por refuerzo
Para la correcta redacción de este apartado del TFG ha sido necesario realizar una revisión del estado del arte del aprendizaje por refuerzo.

i. Proceso de decisión

Un sistema es de Markov si un estado describe el proceso completamente y además el estado es completamente observable.
Estas fórmulas están descritas para espacios discretos de actuación, pero también se aplicarán a espacios continuos.
El método de decisión de Markov es una tupla de la forma $\{\chi , A, T, R, \gamma\}$donde tenemos:
Un conjunto de estados $x \in \chi$
Un conjunto de acciones $a \in A$
Una función de transiciones $T(x, a, x') = p(x'|x, a)$
Una función de recompensas $R(x, a, x')$
La Propiedad de Markov dice que un estado x es de Markov si y sólo si
$p(x_{t+1}|x_t) = p(x_{t+1}|x_1, \dots, x_t)$.
Es decir, dado un estado, no nos importan las acciones anteriores.
Para un estado de Markov (x) y el siguiente (x') la probabilidad de transiciones del estado viene definida por: $P_{x,x'} = p(x_{t+1} = x'|x_t = x)$
La matriz de transición de estado (P) define las probabilidades desde todos los estados (x) a todos los siguientes estados (x')

$$P = \begin{pmatrix} P_{11} & P_{12} & \dots\dots & P_{1n} \\ P_{21} & P_{22} & \dots\dots & P_{2n} \\ \dots\dots & \dots\dots & \dots\dots & \dots\dots \\ P_{n1} & P_{n2} & \dots\dots & P_{nn} \end{pmatrix}$$

ii. Política

Las políticas se asumen que son estocásticas, es decir se dan probabilidades a las acciones por lo que podemos definir la política de la siguiente manera:
Una política π, es una distribución sobre las acciones, dados los estados.

$$\pi(a \mid x) = p(a_t = a \mid x_t = x)$$

La política define por completo el comportamiento de un agente, son "las reglas" que sigue el agente para tomar las acciones.
En los procesos de decisión de Markov las políticas dependen solo del estado actual y no de las acciones pasadas.

iii. Utilidad

La utilidad U es el conjunto de recompensas obtenidas por el agente y si asumimos preferencias estacionarias: $[a_1, a_2, ...] \succ [b_1, b_2, ...] \Rightarrow [r, a_1, a_2, ...] \succ [r, b_1, b_2, ...]$ solo hay una manera de definir la Utilidad: $U([R_0, R_1, R_2, ...]) = R_0 + \gamma \cdot R_1 + \gamma^2 \cdot R_2 + ... +$
Esta utilidad se llama utilidad con descuento ya que $\gamma \in [0, 1)$. Este parámetro de descuento γ, se introduce principalmente para que cuando $\gamma \rightarrow 0$ el agente prefiera obtener recompensas en el presente y no en el futuro. Esto además ayuda a la convergencia matemática, principalmente en problemas de horizonte infinito.

iv. Función de valor y ecuaciones de Bellman

Las funciones de valor estiman cuán positivo será para el agente llegar a un estado.
Hay dos funciones de valor que son las siguientes:
-Función valor: es la utilidad esperada para el agente al estar en un estado (s) y seguir una política(π). $V^{\pi}(x) = E_{\pi}[U_t \mid x_t = x]$
-Función acción-valor: es la utilidad esperada para el agente al estar en un estado (s), tomar una acción (a) y posteriormente seguir la política (π).

$$Q^{\pi}(x, a) = E_{\pi}[U_t \mid x_t = x, a_t = a]$$

Realmente lo que buscaremos será llegar a estados que nos permitan tener buenas funciones valor y acción-valor.
-En las ecuaciones de Bellman para funciones de valor, se parte de la definición y se emplea también la definición de utilidad.

$$V^{\pi}(x) = E_{\pi}[U_t \mid x_t = x]$$
$$V^{\pi}(x) = E_{\pi}[\, R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + ... \mid x_t = x]$$
$$V^{\pi}(x) = E_{\pi}[\, R_{t+1} + \gamma \cdot (R_{t+2} + \gamma \cdot R_{t+3} + ... \mid x_t = x]$$
$$V^{\pi}(x) = E_{\pi}[\, R_{t+1} + \gamma \cdot U_{t+1} \mid x_t = x]$$
$$\rightarrow V^{\pi}(x) = E_{\pi}[\, R_{t+1} + \gamma \cdot V^{\pi}(x_{t+1}) \mid x_t = x]$$

Se nos permite calcular la función valor del estado actual, basándonos en la función valor futura. Se procede de igual manera para la función de acción-valor

$$Q^{\pi}(x, a) = E_{\pi}[U_t \mid x_t = x, a_t = a]$$
$$\rightarrow \quad Q^{\pi}(x, a) = E_{\pi}[\, R_{t+1} + \gamma \cdot Q^{\pi}(x_{t+1}, a_{t+1}) \mid x_t = x, a_t = a]$$

Igualmente, permite calcular la función acción-valor del estado actual, basándonos en la función acción-valor futura.

Además, sabemos que, para espacios discretos se cumple: $E_Z[f(z)] = \sum_{z\in Z} p(z) \cdot f(z)$
Entonces tendremos que la relación entre $V^\pi(x)$ y $Q^\pi(x,a)$ es la siguiente:

$$V^\pi(x) = \sum_{a\in A} \pi(a|x) \cdot Q^\pi(x,a)$$

Las funciones $V^\pi(x)$ y $Q^\pi(x,a)$, en función del estado y acción futuro será de la siguiente manera:

$$\Rightarrow V^\pi(x) = \sum_{a\in A} \pi(a|x) \cdot (\sum_{x'\in X} p(x' \mid x,a) \cdot (R(x,a,x') + \gamma \cdot V^\pi(x')))$$

$$\Rightarrow V^\pi(x) = \sum_{a\in A} \pi(a|x) \cdot Q^\pi(x,a)$$

$$\Rightarrow Q^\pi(x,a) = \sum_{x'\in X} p(x' \mid x,a) \cdot (R(x,a,x') + \gamma \cdot \sum_{a'\in A} \pi(a'|x') \cdot Q^\pi(x',a'))$$

$$\Rightarrow Q^\pi(x,a) = \sum_{x'\in X} p(x' \mid x,a) \cdot (R(x,a,x') + \gamma \cdot V^\pi(x')$$

Esto nos muestra que el valor de $Q(x,a)$ en el estado (x), es igual al valor de la recompensa obtenida tras tomar la acción (a), más el valor de $V(x')$, es decir, la predicción de las recompensas futuras siguiendo la política π

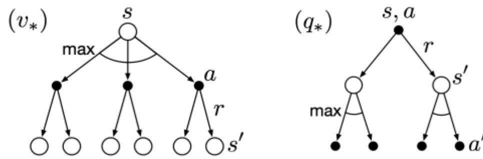
v. Funciones de valor óptimas y política

Se tienen que probar todas las políticas, para obtener las funciones de valor óptimas y escoger entre ellas la mejor.

$V^*(x) = max\{\sum_{x'\in X} p(x'|x,a) \cdot (R(x,a,x') + \gamma \cdot V^*(x'))\}$

$$Q^*(x,a) = \sum_{x'\in X} p(x' \mid x,a) \cdot (R(x,a,x') + \gamma \cdot max\,\{Q^*(x',a')\})$$

Si existe una política óptima nos dará funciones de valor óptimas. Necesitaremos entonces algoritmos para computar $V, Q\ o\ \pi$ para obtener así la política óptima.



-Ecuaciones de Bellman para funciones de valor óptimas:

$$V^*(x) = max\ Q^*(x, a)$$

$$Q^*(x, a) = \sum_{x' \in X} p(x' \mid x, a) \cdot (R(x, a, x') + \gamma \cdot V^*(x'))$$

$$\Rightarrow V^*(x) = max\ (\sum_{x' \in X} p(x' \mid x, a) \cdot (R(x, a, x') + \gamma \cdot V^*(x')))$$

$$\Rightarrow Q^*(x, a) = \sum_{x' \in X} p(x' \mid x, a) \cdot (R(x, a, x') + \gamma \cdot max\ Q^*(x', a'))$$

b. Herramientas de predicción

La Evaluación de Montecarlo y la Diferencia Temporal son herramientas necesarias para calcular las funciones de valor $V^\pi(x)$ y $Q^\pi(x, a)$, es decir, estimar recompensas futuras.

i. Evaluación de Montecarlo

El objetivo de la evaluación de Montecarlo es aprender V^π a partir de los episodios de experiencia bajo la política π. $V^\pi(x) = \sum_i U_t^{(i)} 1_{x_t^{(i)} = x} \quad \forall i \in episodes$

Esto quiere decir que la función de valor es el sumatorio de todas las soluciones. El 1 es una función indicadora que actúa como un if informático, será 1 si $x_t^{(i)} = x$ y 0 si $x_t^{(i)} \neq x$.

siendo: $U_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots R_n$

$V^\pi(x) = E_\pi[U_t \mid x_t = x]$

Teniendo en cuenta que $E(A) = \tilde{A}_N \approx \frac{1}{N} \cdot \sum_{i=1}^{N} a_i = \tilde{A}_{N-1} + \frac{1}{N} \cdot (a_N - \tilde{A}_{N-1})$, el valor medio de N número de datos es el valor medio calculado en el dato anterior más la novedad entre el número de puntos, siendo la novedad el valor nuevo menos el valor medio en el dato anterior.

Con ayuda de esta última expresión tenemos que la manera de implementar la evaluación de Montecarlo incremental es: $V(x_t) \leftarrow V(x_t) + \frac{1}{N(x_t)} \cdot (U_t - V(x_t))$

con: $N(x_t) \leftarrow N(x_t) + 1$

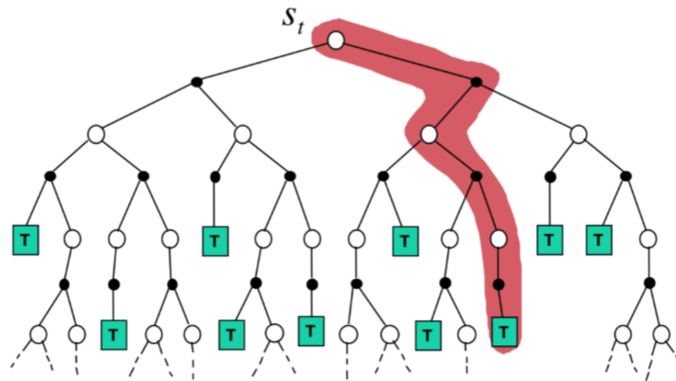
La actualización de la función de valor, que es lo que nos interesa para más adelante, será de la siguiente forma:

$$\Rightarrow V(x_t) \leftarrow V(x_t) + \alpha \cdot (U_t - V(x_t))$$

$U_t$es la recompensa recogida durante el episodio
$V(x_t)$es la recompensa media recogida en los episodios previos

Más gráficamente lo vemos de la siguiente manera:

$$V(x_t) \leftarrow V(x_t) + \alpha(U_t - V(x_t))$$



En la evaluación de Montecarlo se tienen que terminar los episodios completamente y esto es un problema. Además, no se puede aplicar para sistemas continuos e ignora la estructura MDP.
Los problemas que presenta la evaluación de Montecarlo se tratan de solventar con la diferencia temporal (DT).

ii. Diferencia Temporal

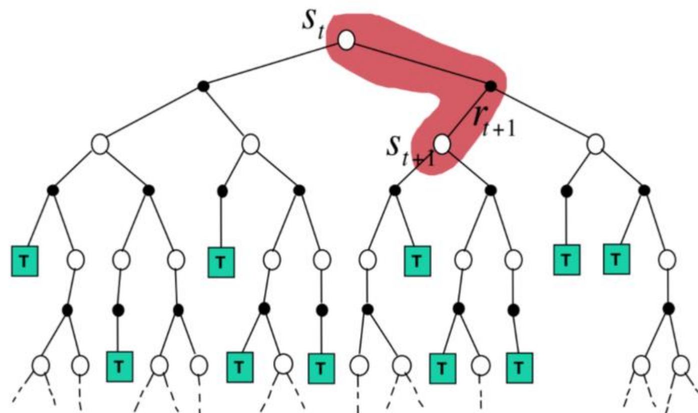
El objetivo de la DT es aprender de cualquier interacción, por lo que no es necesario ejecutar los episodios por completo sino dar un paso y emplear la función de valor para estimar el comportamiento futuro.
Las ventajas que presenta la DT son que sirve para sistemas continuos, puede aprender de secuencias incompletas y explota la estructura MDP.
La forma de implementar función de valor con la diferencia temporal es la siguiente:

$$\Rightarrow V(x_t) \leftarrow V(x_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(x_{t+1}) - V(x_t)) = V(x_t) + \alpha \ \cdot \ error$$

$\alpha \rightarrow 1$ Se emplea para olvidar el pasado lejano.
$\gamma \rightarrow 0$ Se emplea para ignorar el futuro lejano.

La forma gráfica es la siguiente:

$$V(x_t) \leftarrow V(x_t) + \alpha(R_{t+1} + \gamma V(x_{t+1}) - V(x_t))$$



c. Optimización de la política.

i. Basado en la función de valor

La optimización de la política mediante la función Q es la técnica que emplean algunos de los algoritmos estudiados.
Computar el "greedy"$V(x)$ sería imposible, ya que como hemos dicho anteriormente, en el aprendizaje por refuerzo sin modelo de primeras no se conoce ningún dato. Por lo tanto no podemos calcular la probabilidad $p(x'|x,a)$, ni conocer la recompensa $R(x,a,x')$, lo cual es necesario para obtener la política óptima:

$$\pi^*(x) = argmax \quad V(x) = argmax\{\sum_{x\prime \in X} \quad p(x'|x,a) \cdot (R(x,a,x') + \gamma \cdot V^*(x'))$$

Como solución se computar la política greedy de $Q(x,a)$ya que con la función Q no necesitaremos la función de transiciones del primer paso:

$$\pi^*(x) = argmax \quad Q(x,a)$$

La diferencia temporal para la función de valor Q off-Policy es de la siguiente manera:

$$Q(x_t,a_t) \leftarrow Q(x_t,a_t) + \alpha \cdot (R_{t+1} + \gamma \cdot max(Q(x_{t+1},a') - Q(x_t,a_t)))$$

En el Q-Learning hay una política $\pi\ (e - greedy)$ para las acciones que se toman a_t y otra política $\mu\ (greedy)$para las acciones futuras a'.
En realidad, la forma de modelar la función Q es empleando redes neuronales profundas ya que para problemas complejos sería imposible guardar todos los estados posibles en tablas.

ii. Basado en la búsqueda de la política

En el método de búsqueda de la política (Policy Search) en vez de emplear el valor Q se va a abordar directamente el problema de optimizar la política $\pi(x,a)$.
Se aprenden los parámetros de la función directamente mediante la interacción.
Para cualquier función $f(\cdot)$: $E_x[f(x)] = \sum_x \quad f(x) \cdot p(x)$

$$\nabla_\theta \, logf(\theta) = \frac{\nabla_\theta \, f(\theta)}{f(\theta)} \Rightarrow \nabla_\theta \, f(\theta) = f(\theta) \cdot \nabla_\theta \, logf(\theta)$$

Dada una política $\pi_\theta(x,a)$ se aprenden los parámetros de las funciones.
Para cuantificar cómo de buenos son los parámetros necesitamos definir la función de coste $J(\theta)$ tratando de maximizarla mediante ascenso de gradientes para obtener así la máxima recompensa posible.

$$J(\theta) = E_p \cdot [\sum_{a_{0:T}} \quad \cdot (\sum_{t=0}^{T} \quad R(x_t, a_t, x_{t+1})) \cdot \pi_\theta(a_{0:T} \mid x_{0:T})]$$

Aplicando el truco del gradiente del logaritmo de una función tenemos lo siguiente: $\nabla_\theta \, J(\theta) = E_\tau[\sum_{t=0}^{T} \quad (U_t \cdot \nabla_\theta \, log\pi_\theta(a_t \mid x_t)]$
Además, podremos hacer una aproximación usando Montecarlo para N episodios:

$$\nabla_\theta \, J(\theta) \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \quad \sum_{t=0}^{T} \quad U_t^{(i)} \cdot \nabla_\theta \, log\pi_\theta(a_t^{(i)} \mid x_t^{(i)})$$

Lo importante de esta aproximación es que con $N = 1$ tenemos una buena aproximación, por lo tanto, con una muestra será suficiente.
$\widehat{\nabla_\theta \, J(\theta)} \approx \frac{1}{N} \cdot \sum_{i=1}^{N} \quad \nabla_\theta \, J_i(\theta)$
Los parámetros se actualizarán de la siguiente manera: $\theta \approx \theta - \alpha \cdot \nabla_\theta \, J_i(\theta)$, teniendo que evaluar un episodio diferente cada vez. Esta forma de operar se llama Stochastic Gradient Descent y se muestra a continuación de manera gráfica.