Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

Reinforcement learning como reacción frente a anomalías en la red

Máster en Ingeniería de Telecomunicación

Autor:
GUILLERMO CAMINERO FERNÁNDEZ

Tutor:
BELÉN CARRO MARTÍNEZ

Valladolid, 27 de junio de 2018

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Resumen

Teoría de la Señal y Comunicaciones e Ingeniería Telemática

Máster en Ingeniería de Telecomunicación

Reinforcement learning como reacción frente a anomalías en la red

por GUILLERMO CAMINERO FERNÁNDEZ

Los algoritmos de aprendizaje reforzado o reinforcement learning son un tipo de algoritmos de machine learning que permiten a los agentes software determinar automáticamente el comportamiento ideal en un determinado contexto, con el objetivo de maximizar una recompensa mediante prueba y error. Se pretende evaluar la adecuación de reinforcement learning como método de reacción automática frente a un determinado tipo de problemas que puedan surgir en la red, relacionados con anomalías de seguridad, bien de manera aislada o en conjunción con otras técnicas de deep learning.

Palabras Clave: Reinforcement learning, aprendizaje reforzado, aprendizaje automático, ANN, detección de anomalías, KDD'99, ciberseguridad

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

Abstract

Teoría de la Señal y Comunicaciones e Ingeniería Telemática

Master's Degree in Telecommunication Engineering

Reinforcement learning as a reaction against network anomalies

by GUILLERMO CAMINERO FERNÁNDEZ

Reinforced learning algorithms are a class of machine learning algorithm that allow software agents to automatically determine the optimal behavior in a given context, in order to maximize a reward through trial and error. The aim is to evaluate the suitability of reinforcement learning as a method of automatic reaction to a certain type of problem that may arise on the network, related to security anomalies, either in a single way or in conjunction with other deep learning techniques.

Keywords:Reinforcement learning, machine learning, ANN, anomaly detection, KDD'99, cybersecurity

Agradecimientos

Quisiera dedicar este apartado a todas aquellas personas que se han prestado para hacer posible la realización de este trabajo fin de máster.

Cabe destacar la ayuda de la tutora del trabajo por el apoyo dado, además de la orientación y seguimiento durante los meses de la realización del trabajo.

De igual manera, mi agradecimiento a Manuel Lopez Martín por su apoyo y ayuda. Siempre ha contestado rápido y amable a las dudas que me han ido surgiendo.

Por otro lado, y no menos importante, me gustaría dar las gracias a mi familia por su gran apoyo y cariño durante todos los años del recorrido académico y en la realización del proyecto. También quisiera agradecer el apoyo de mi novia por su paciencia y ánimos durante toda la realización del trabajo.

Gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Resumen de notación

$\doteq$	Relacción de igualdad que es verdadera por definición
$\approx$	Aproximadamente igual
$\mathbb{E}[X]$	Esperanza de la variable aleatória X
$\mathcal{P}\{X = x\}$	Probabilidad de que la variable aleatória X tome el valor x
$\text{argmax}_a\, f(a)$	Valor de a para el que la función $f(a)$ obtiene su valor máximo
$\log(x)$	Logaritmo natural de x
$\exp(x)$	e^x, donde $e \approx 2{,}71828$ es la base del logaritmo natural; $\exp(\ln(x)) = x$
$\mathbb{R}$	Conjunto de los números reales
ϵ	Probabilidad de tomar una acción aleatoria en una política ϵ-greedy
α, β	Parámetros del tamaño de paso para los algoritmos
γ	Parámetro de descuento
λ	Parámetro de decaimiento

En los problemas de optimización *multi-arm bandit*:

k	Número de acciones
t	Paso de tiempo discreto
$q_*(a)$	Valor real (recompensa esperada) de la acción a
$\pi_t(a)$	Probabilidad de seleccionar la acción a en el instante de tiempo t

En un Proceso de Decisión de Markov:

s, s'	Estados
a	Acción
r	Recompensa
$\mathcal{S}$	Conjunto de estados no terminales
$\mathcal{S}^+$	Conjunto de todos los estados, incluyendo los terminales
$\mathcal{A}$	Conjunto de todas las acciones posibles
$\mathcal{R}$	Conjunto de todas las posibles recompensas, subconjunto de $\mathbb{R}$
$\subset$	Subconjunto de; p.ej., $\mathcal{R} \subset \mathbb{R}$
$\in$	Elemento perteneciente a; p.ej.,$s \in \mathcal{S}, r \in \mathcal{R}$
t	Paso de tiempo discreto
$T, T(t)$	Paso de tiempo final de un episodio, o final incluyendo el paso t
A_t	Acción en el paso t
S_t	Estado en el paso t
R_t	Recompensa en el paso t
π	Política
$\pi(s)$	Acción tomada en el estado s bajo la política *determinista* π
$\pi(a \mid s)$	Probabilidad de tomar la acción a en el estado s bajo la política *estocástica* π
$\pi(a \mid s, \boldsymbol{\theta})$	Probabilidad de tomar la acción a en el estado s y el vector de parámetros $\boldsymbol{\theta}$
G_t	Recompensa acumulativa descontada después de un tiempo t
$p(s', r \mid s, a)$	Probabilidad de que suceda la transición a s' con una recompensa r, estando en el estado s y tomando la acción a

$p(s' \mid s, a)$	Probabilidad de que suceda la transición a s', estando en el estado s y tomando la acción a
$v_\pi(s)$	Valor del estado s bajo la política π (recompensa esperada)
$v_*(s)$	Valor del estado s bajo la política óptima
$q_\pi(s, a)$	Valor de tomar la acción a en el estado s bajo la política π
$q_*(s, a)$	Valor de tomar la acción a en el estado s bajo la política óptima

Para *Policy Gradient*:

$\mathbf{w}$	Vector de pesos de tamaño d que aproximan una función de valor
d	Número de componentes del vector $\mathbf{w}$
d'	Número de componentes del vector $\boldsymbol{\theta}$
$\hat{v}(s, \mathbf{w})$	Valor aproximado dado el estado s y el vector de pesos $\mathbf{w}$
$\hat{q}(s, a, \mathbf{w})$	Valor aproximado del par estado-acción dado el vector de pesos $\mathbf{w}$
$\boldsymbol{\theta}, \boldsymbol{\theta}_t$	Vector de parámetros de la política
$\pi_{\boldsymbol{\theta}}$	Política correspondiente con el vector de parámetros $\boldsymbol{\theta}$
$J(\pi), J(\boldsymbol{\theta})$	Rendimiento de la medida para la política π o $\pi_{\boldsymbol{\theta}}$

Capítulo 1

Introducción, motivación y objetivos

1.1. Introducción

En los últimos años la preocupación por la ciberseguridad ha crecido de forma exponencial. La aparición de dispositivos *IoT*, el aumento del número de dispositivos personales, así como el aumento de la cantidad de datos privados que existen de forma alcanzable a través de la red, hacen que la ciberseguridad sea a día de hoy uno de los temas más estudiados y temidos. Al mismo tiempo que se desarrollan nuevas técnicas de detección de intrusos o de ataques, aparecen nuevos tipos de ataques más sofisticados. Este proceso de evolución de los ataques, necesita de mecanismos que se puedan adaptar de forma inteligente a los requerimientos actuales de las redes y los dispositivos. La mayoría de los ataques que se efectúan son ligeras modificaciones de otros que ya se conocen. Estas modificaciones se pueden detectar aprovechando el conocimiento de la firma del anterior. El problema aparece cuando los ataques son completamente nuevos: para estos casos, se necesitan sistemas novedosos que consigan generalizar y aprender en el momento del ataque y, además, haciendo posible una reacción ante el ataque. En los últimos años, la inteligencia artificial está adquiriendo mucha fuerza, dada la potencia de procesamiento de los ordenadores actuales y el gran interés que se tiene por avanzar en este tema. La inteligencia artificial, desgraciadamente, se puede utilizar tanto para la detección de ataques como para la generación de nuevos. En este ámbito aparece el interés de ir un paso por delante de los posibles ataques inteligentes y el desarrollo de nuevas técnicas que permitan mitigarlos. Es posible que en un futuro no muy lejano los ataques informáticos sean lanzados completamente por máquinas dotadas de inteligencia artificial, por lo que las defensas deben estar provistas de mecanismos similares deseablemente más inteligentes y sin la necesidad de la supervisión humana. Por este motivo, se analizan diferentes posibilidades de la detección de estas anomalías utilizando la teoría de *reinforcement learning*.

La estructura general de este documento se presenta de la siguiente forma: en el capítulo 2 se detallan en profundidad los aspectos necesarios sobre la teoría de *Reinforcement Learning*, que será aplicada en los detectores. Una vez explicada la teoría necesaria sobre *RL*, se detalla el proceso completo para la implementación del detector de anomalías. Este proceso se explica a lo largo del capítulo 3 donde se muestran todos los pasos que se siguen desde que se escogen los datos, se seleccionan las características, se procesan dichos datos y se crea el detector. En este mismo capítulo, se pueden encontrar tres distintas alternativas de detectores: un detector simple que se encarga de detectar de forma binaria los ataques, otro detector más complejo, que detecta específicamente el ataque que se ha llevado a cabo y, por último, uno que detecta el tipo de ataque que se ha producido. Una vez se muestran los tipos de detectores se pasa a

ofrecer los resultados que se consigue con ellos. Estos detectores utilizan la teoría básica de *RL* que permite su correcto funcionamiento, aunque los resultados se puedan mejorar. Este es el objetivo que se describe en el capítulo 5 en el que se procede a seguir los pasos de la propia evolución que ha seguido a lo largo de los años la teoría de los algoritmos de *RL*. Por último en el capítulo 6 se detalla una pequeña comparación entre los distintos algoritmos utilizados, así como las posibles mejoras o lineas futuras.

1.2. Motivación

La motivación principal de este trabajo es poder aplicar una tecnología tan novedosa como *reinforcement learning* a un tema tan importante como el de la ciberseguridad. Este método está en completo auge por sus capacidades de descubrimiento y aprendizaje de acciones sin estar precisamente etiquetadas como hace un método supervisado. Existen multitud de campos en el que estas técnicas están siendo estudiadas y cada día que pasa este abanico aumenta. Posiblemente una de las aplicaciones más recientes en este ámbito y más mediática puede ser el desarrollo de **AlphaGo** [1]. En esta aplicación, un algoritmo sin ningún tipo de conocimiento del juego *Go*, fue capaz de vencer a los mejores jugadores del mismo, considerándose este juego el más complejo de todos. Este paso significó un gran avance e interés en la materia y desde entonces esta tecnología ha sido extensamente utilizada e investigada. La motivación inicial por la técnica *RL*, sumada al importante campo de la ciberseguridad, hacen del tema de estudio un interesante punto de investigación.

1.3. Objetivos

Los objetivos de este proyecto son claros y se dirigen con una única finalidad: obtener el mejor detector de anomalías posible utilizando la técnica de *Reinforcement Learning*. El objetivo de un buen detector no es solo poder detectar con precisión las muestras que se le ofrecen en el conjunto de datos, sino que un buen detector tiene que ser capaz de generalizar y adaptarse a nuevas posibles muestras derivadas y completamente nuevas amenazas.

Capítulo 2

Reinforcement Learning

2.1. Introducción a RL

Reinforcement Learning o el aprendizaje por refuerzo es una forma de aprendizaje automático, por la que una agente busca entre las acciones óptimas que puede realizar con el fin de que se maximice una recompensa. Este tipo de aprendizaje está profundamente arraigado en las perspectivas psicológicas y neurocientíficas sobre el comportamiento animal. El "aprendiz" no tiene noción de las acciones que tiene que tomar, pero debe descubrir cual de ellas maximizan no solo la recompensa inmediata sino que una recompensa a largo plazo. Las características de prueba-error y búsqueda de la recompensa retardada, son dos de los aspectos más importantes del aprendizaje por refuerzo [2].

Se puede considerar el aprendizaje por refuerzo en el centro de muchas áreas de la ciencia que se relacionan entre ellas. Para ello el aprendizaje por refuerzo simplemente muestra la forma de efectuar un control óptimo [3]. Esta relación se muestra en la Figura 2.1. Todas las áreas del conocimiento se pueden solucionar con la teoría de la toma de decisiones.

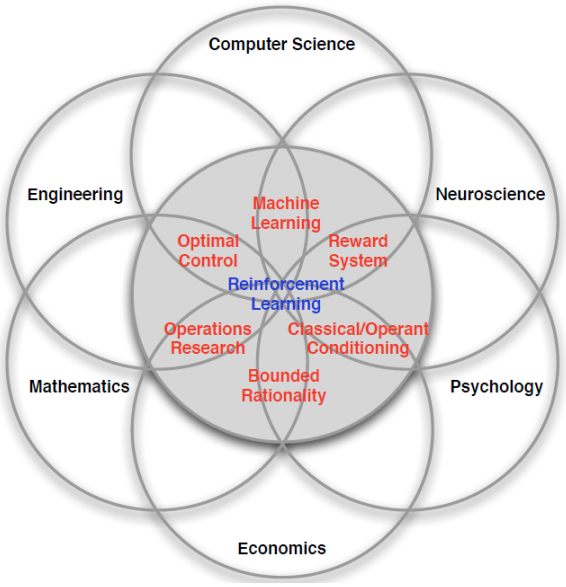

FIGURA 2.1: Caras del aprendizaje por refuerzo [3]

Dentro de los dos pilares del aprendizaje máquina o *machine learning* (aprendizaje supervisado y no supervisado), *Reinforcement Learning* se separa de ambos por poseer ciertas características que no se podrían clasificar en ninguno de ellos.

El **aprendizaje supervisado** aprende de un conjunto de datos previamente etiquetados. Cada ejemplo se compone de un conjunto de características o *features* que se corresponden con una descripción del propio ejemplo junto con una etiqueta o *labels* de la acción correcta que debe tomar el sistema. El objetivo de este aprendizaje es que el sistema generalice sus respuestas para que actúe correctamente en situaciones no presentes en el conjunto de datos. Este tipo de aprendizaje es muy importante y utilizado, pero no es adecuado para que el algoritmo aprenda por si mismo únicamente de las interacciones con el entorno. En un territorio desconocido, donde se necesitaría un aprendizaje óptimo, un agente debe de ser capaz de aprender de su propia experiencia. Este es aspecto fundamental que diferencia el aprendizaje reforzado del supervisado, y es que en el aprendizaje reforzado no hay un supervisor, se basa únicamente en el aprendizaje de prueba y error.

El aprendizaje **no supervisado**, por otro lado, se basa en crear una estructura oculta en una colección de datos no etiquetados.

A simple vista, los términos de aprendizaje supervisado y aprendizaje no supervisado parecen clasificar exhaustivamente los paradigmas de aprendizaje automático, pero esto no es así. Se puede prensar que el aprendizaje por refuerzo entra dentro del tipo de aprendizaje no supervisado, porque no se basa en ejemplos de comportamiento correcto o etiquetados. Sin embargo, el aprendizaje por refuerzo trata de maximizar la señal de recompensa en lugar de encontrar una estructura oculta en los datos. Descubrir la estructura de la experiencia de un agente puede ser útil en el aprendizaje por refuerzo, pero por si mismo no aborda el problema de maximizar la señal de recompensa.
Otro de los aspectos que diferencian el aprendizaje reforzado del aprendizaje supervisado es que éste último obtiene sus resultados de forma instantánea, mientras que en el aprendizaje reforzado, las recompensas se ofrecen de forma retardada (no siempre la mejor recompensa actual es la más apropiada para el futuro). También hay que tener en cuenta la temporalidad de los eventos. En muchos casos, en el aprendizaje supervisado o no supervisado, se presentan los datos como datos incorrelados o *IID*. Esto no sucede normalmente en el aprendizaje por refuerzo ya que el tiempo en el que aparecen los sucesos es un dato muy importante a la hora de ofrecer las recompensas.

Por todo lo comentado, se considera el aprendizaje por refuerzo un tercer paradigma del aprendizaje automático, tal y como se muestra en la figura 2.2 junto con el aprendizaje supervisado y el aprendizaje no supervisado.

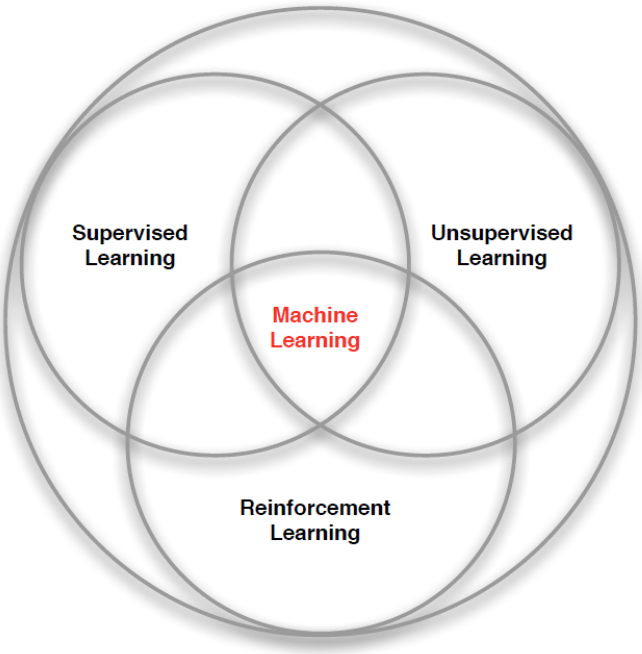

FIGURA 2.2: Secciones del aprendizaje automático [3]

Uno de los desafíos en el *reinforcement learning*, y no presente en el resto de tipos de aprendizaje, es el equilibrio entre la exploración y la explotación. Para obtener unas recompensas elevadas, el agente del *RL* debe preferir acciones que ha probado en el pasado y que efectivamente han producido altas recompensas. Pero para descubrir estas acciones, es necesario probar acciones que no se han seleccionado con anterioridad. El agente tiene que **explotar** el conocimiento actual que posee en orden de obtener recompensas, pero también tiene que **explorar** para descubrir posibles mejores acciones en el futuro. El dilema es, que ni la exploración ni la explotación pueden llevarse a cabo exclusivamente sin fracasar en la tarea. El agente debe intentar una variedad de acciones y favorecer progresivamente las que "aparentan" ser las mejores. En una actividad estocástica, cada acción tiene que ser probada muchas veces para obtener una estimación fiable de la recompensa esperada. El dilema de la **exploración-explotación** ha sido intensamente estudiado por matemáticos durante muchas décadas, pero sigue sin resolverse [2].

2.1.1. Elementos en un problema de *RL*

Los elementos principales que se pueden considerar en un problema de *RL* son el agente y el entorno. Estos dos interaccionan directamente, de forma que el agente trata de maximizar la recompensa obtenida y el entorno representa las reglas en las que se mueve el agente. De este modo el entorno responde a las acciones originadas por el agente ofreciendo las recompensas y las observaciones en las que se encuentra actualmente. Estas relaciones se pueden observar de forma esquemática en la figura 2.3.

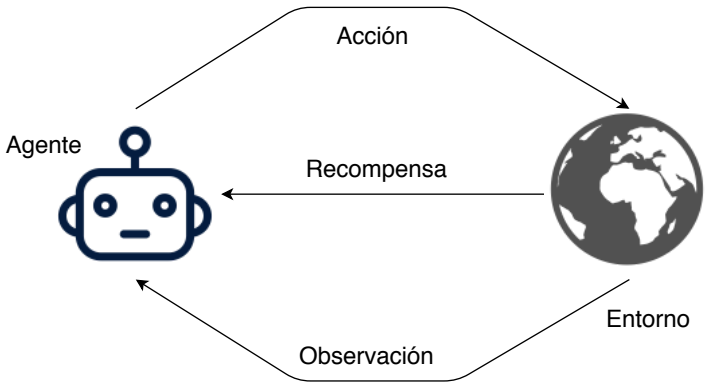

FIGURA 2.3: Descripción del agente y el entorno

Más allá del agente y el entorno, se pueden identificar cuatro subelementos de un sistema de aprendizaje por refuerzo: una política, una señal de recompensa, una función de valor y, opcionalmente, un modelo del entorno [2].

Política

Define el comportamiento del agente en un momento dado. En términos generales, una política es un mapeo desde los estados percibidos del medio a las acciones que se deben tomar en esos estados. Corresponde a lo que en psicología se llamaría un conjunto de reglas o asociaciones de estímulo-respuesta. En algunos casos la política puede ser una simple función o tabla, mientras que en otros, puede implicar un cálculo extensivo como un proceso de búsqueda. La política es el núcleo de un agente de *RL*, en el sentido de que por sí sola es suficiente para determinar el comportamiento. En general, las políticas suelen ser estocásticas.

Dado un *MDP* (apartado 2.2), una política es una función que ofrece como salida una acción $a \in A(s)$ para cada estado $s \in S$. Formalmente, una política determinista π es una función definida como $\pi : S \times A \to [0, 1]$ de tal manera que para cada estado $s \in S$, sostiene que $\pi(s, a) \geq 0$ y $\sum_{a \in A} \pi(s, a) = 1$ [4]. Por otro lado, una política estocástica sostiene que $\pi(a \mid s) = \mathcal{P}[A_t = a \mid S_t = s]$.

Recompensa

Es el componente de *RL* que define la meta del aprendizaje. Se denomina como R_t, y en cada instante temporal, el entorno la envía al agente como un valor escalar. Indica el cómo de bien lo está haciendo el agente en el instante temporal t. El objetivo del agente es maximizar a largo plazo el valor total de estas recompensas. La señal de recompensa define de este modo cuales son los eventos buenos o malos para el agente y es la base primaria para alterar la política. Si una acción seleccionada por la política obtiene una recompensa baja, entonces la política se modifica para seleccionar alguna otra acción que mejore la recompensa en el futuro. En general, existen distintos tipos de definiciones para la recompensa. La más habitual es la dependiente del estado, de forma que $R : S \to \mathbb{R}$. Pero también se pueden definir dependiendo del estado y la acción elegida $R : S \times A \to \mathbb{R}$ o incluso si se tiene en cuenta el estado destino $R : S \times A \times S' \to \mathbb{R}$.

Función de valor o *value function*

Indica el valor de estar en un estado con previsión a largo plazo. En términos generales, el valor de un estado es la cantidad total de recompensa que el agente puede esperar acumular en el futuro a partir de ese estado. Al contrario que las recompensas que determinan la conveniencia inmediata e intrínseca de los estados, los valores indican la conveniencia a largo plazo teniendo en cuenta que atravesará los estados más probables y recibirá las recompensas asociadas a ese camino. Las recompensas son primarias y los valores son en cierto sentido secundarias ya que no existe función de valor sin la recompensa. El único propósito de estimar la función de valor es lograr más recompensas en el futuro. Las decisiones de elegir una acción u otra se toman en base a juicios de valor. Se buscan, por lo tanto, las acciones que produzcan mayor valor y no mayor recompensa ya que se trata de maximizar la recompensa a largo plazo. Desafortunadamente es más difícil determinar el valor que las recompensas. Las recompensas las ofrece el medio, mientras que los valores tienen que ser estimados con cada secuencia de observaciones que hace el agente a lo largo de su existencia. Por lo tanto, el componente más importante de casi todos los algoritmos de *RL* es el método de estimación eficiente de la función de valor [2].

Modelo del entorno

Corresponde con el elemento que imita el comportamiento del medio ambiente, o más generalmente, permite hacer inferencias sobre cómo se comportará el medio ambiente. Los métodos para resolver problemas de *RL* que utilizan modelos y planificación se denominan métodos basados en modelos, a diferencia de los métodos más sencillos sin modelos que son explícitamente experimentos de prueba y error.

Estado, acción y probabilidad de transición

No específicamente como elementos de *RL* pero necesaria su definición. Se define el conjunto de estados S como un conjunto finito $\{s_1, s_2, ..., s_N\}$ de tamaño N. Un estado corresponde con la caracterización única de todo lo que es importante en un momento dado del problema que se modela.

El conjunto de acciones A se define como el conjunto finito $\{a_1, a_2, ..., a_K\}$ de tamaño total K. El conjunto de acciones que pueden ser aplicadas en algún estado en particular $s \in S$, se denomina $A(s)$, donde $A(s) \subseteq A$. En algunos sistemas, no todas las acciones pueden ser aplicadas en todos los estados, pero en general se asume que $A(s) = A$ para todos los $s \in S$.

Por último, la probabilidad de transición o función de transición, representa la distribución de probabilidad sobre el conjunto de las posibles transiciones que efectúan el cambio de un estado s a uno s' tomando la acción a.

2.2. Markov Decision Processes

Los procesos de decisión de Markov o *MDP* describen perfectamente los entornos del aprendizaje reforzado. Teniendo en cuenta los elementos definidos con anterioridad, si un proceso es de Markov, dicho proceso se define completamente en el estado actual en el que se encuentra. Los procesos de decisión de Markov están pensados para ofrecer un marco sencillo para el aprendizaje mediante la interacción. El agente interactúa continuamente escogiendo acciones y el entorno responde a estas acciones ofreciendo los estados resultantes o las nuevas situaciones y las recompensas para cada situación. Esta interacción se aprecia de forma clara en el esquema de la figura 2.4. Más específicamente, el entorno interactúa en los instantes de tiempo discretos[1] $t = 0, 1, 2, 3, \ldots$. En cada instante temporal t, el agente recibe una representación del entorno o lo que se conoce como el *estado* del entorno $S_t \in S$. En base a este estado, el agente selecciona una *acción* $A_t \in A(s)$. Un instante temporal después, en consecuencia de la acción tomada, el agente recibe una *recompensa* $R_{t+1} \in R \subset \mathbb{R}$ y encuentra un nuevo estado S_{t+1}. El *MDP* y el agente dan lugar a una secuencia o trayectoria del estilo a la siguiente:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.1}$$

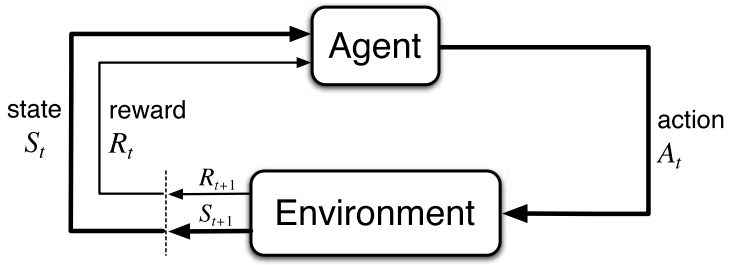

FIGURA 2.4: Interacción entre el agente y el entorno en un *MDP* [2]

En un *MDP* finito, el conjunto de estados, acciones y recompensas ($\mathcal{S}$, $\mathcal{A}$ y $\mathcal{R}$) contienen un número finito de elementos. En este caso, las variables aleatorias R_t y S_t tienen bien definidas una distribución de probabilidad dependiente únicamente del estado y acción que les precede. Esta probabilidad se define como la probabilidad de transición y que para un *MDP* se simplifica de acuerdo a la siguiente ecuación:

$$p(s', r \mid s, a) \doteq \mathcal{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a) \tag{2.2}$$

[1]Se refiere a instantes de tiempo discretos con el fin de simplificar la descripción de los *MDP*, no obstante, se puede extender al dominio continuo realizando las modificaciones oportunas.

La función de probabilidad de transición debe cumplir también la siguiente ecuación en función de la probabilidad unidad:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s' \mid s, a) = 1, \text{Para todo } s \in \mathcal{S},\ a \in \mathcal{A}(s) \tag{2.3}$$

De esta forma se caracteriza completamente la dinámica de un *MDP* finito con la función $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. A partir de esta función se puede calcular la probabilidad de transicción de estado o *state-transition probabilities*. En este caso se denota como $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ y se define como la siguiente ecuación:

$$p(s' \mid s, a) \doteq \mathcal{P}(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{2.4}$$

Como se ha comentado con anterioridad, la recompensa en el instante temporal t queda definida por el estado y la acción tomada en $t - 1$. Teniendo en cuenta esto, se puede expresar la recompensa media dependiente del par estado-acción como una función de dos argumentos $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Esta ecuación se puede definir como la suma de todas las recompensas por la probabilidad asociada a cada transición de estado.

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \tag{2.5}$$

Por otro lado, se puede expresar la recompensa esperada como función de tres términos correspondientes al estado, acción y siguiente estado $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. De este modo, la función de recompensa queda totalmente caracterizada para el entorno en función de las probabilidades de transición. Esto posibilita el encontrar un máximo de esta función en términos conocidos.

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)} \tag{2.6}$$

El marco de un *MDP* se corresponde con la abstracción de un problema orientado a objetivos a partir de la interacción. Propone que sean cuales sean los encargados del aparato sensorial, de memoria, control; y cualquiera que sea el objetivo que se pretende alcanzar, se puede lograr con un aprendizaje orientado a metas. Este puede reducirse a tres señales que pasan de un agente a su entorno. Una señal para representar las elecciones hechas por el agente (las acciones), una señal para representar la base sobre la cual se hacen las elecciones (los estados), y una señal para definir la meta del agente (las recompensas) [2].

2.2.1. *Discounted rewards*

Dado que los sistemas de *RL* son orientados a metas, la cuestión es cómo definir las mismas. La forma más sencilla, teniendo en cuenta las señales devueltas por el entorno, es sumar todas las recompensas esperadas para obtener la secuencia de recompensas G_t (*Goal* o meta) donde T representa el instante de tiempo final.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.7}$$

Una vez definida la función de recompensa total, se puede aplicar a tareas con interacciones entorno-agente que se expresan en forma de **episodios**. Estos episodios deben terminar en un estado final que se denomina **estado terminal** (*terminal state*) seguido de un *reset* o un estado sin fin en el que no se ofrecen nuevas recompensas. Este tipo de tareas se consideran **episódicas** y se debe identificar todos aquellos estados que sean no terminales denotados como $\mathcal{S}$ y los terminales $\mathcal{S}^+$. La longitud de intervalos T no siempre es constante y puede variar a lo largo de los episodios.
Por otro lado, en muchos casos, las tareas no se pueden segmentar en episodios ya que continúan indefinidamente en el tiempo. En este caso se puede definir $T = \infty$ y cuando se pretenda maximizar la recompensa total esta tarea, probablemente sume infinito. Por este motivo se utilizan recompensas descontadas, que se pueden entender como un beneficio de obtener una recompensa en el momento actual que tiene que ver con todos los instantes hacia el futuro. No obstante un agente siempre podrá escoger una menor recompensa en el instante actual, siempre y cuando maximice el total a lo largo de su existencia. Para definir este tipo de recompensas se utiliza el parámetro *discount rate* o tasa de descuento γ donde $0 \leq \gamma \leq 1$. La ecuación que se corresponde con la recompensa descontada total se define como la siguiente:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.8}$$

En este punto, se puede abstraer dicha ecuación con una relación de recursividad y será realmente útil en los algoritmos de aprendizaje reforzado:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.9}$$

Es importante remarcar que esta forma de definir las recompensas descontadas asegura que la recompensa total esperada sea un número finito incluso si el número de episodios es infinito.

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{2.10}$$

2.3. Políticas y *Value functions*

Casi todos los algoritmos de aprendizaje reforzado implican la estimación de las funciones de valor o *value functions* que son funciones dependientes del estado o del estado-acción. Esta *value function* estima como de bueno es para el agente, estar en un determinado estado. El como de bueno, en este caso, viene determinado en términos de recompensas futuras que pueden obtener partiendo de ese estado. En consecuencia, las *value functions* se definen con respecto a formas particulares de actuar, llamadas políticas.

Formalmente, una *política* π es un mapeo desde los estados hasta las probabilidades de seleccionar cada acción posible. Si el agente sigue la política π en el instante de tiempo t, entonces $\pi(a \mid s)$ es la probabilidad de que $A_t = a$ si $S_t = s$. Existen distintas formas de especificar cómo cambia la política del agente con la experiencia [2].

El valor de un estado o *state-value* bajo la política π, se denota como $v_\pi(s)$ y corresponde con el valor esperado de recompensas si se empieza desde el estado s y se sigue la política π. Para un *MDP*, se define formalmente el valor de $v_\pi(s)$ como:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{2.11}$$

Esta ecuación se cumple para todo $s \in \mathcal{S}$. Además hay que tener en cuenta que a partir del estado terminal (si es que lo hay), todas las recompensas son cero. Similarmente, se puede definir el valor de tomar la acción a en el estado s bajo la política π. Este modo de definir el valor se conoce como *action-value function* para la política π, se expresa como $q_\pi(s, a)$ y se define en la siguiente ecuación:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.12}$$

Las funciones de *valor* pueden ser estimadas por medio de la experiencia. Por ejemplo si un agente sigue un política π y mantiene un valor medio para cada valor de estado encontrado, cuando el número de muestras tiendan a infinito, el valor medio converge al *state value* $v_\pi(s)$. Del mismo modo se puede llegar al valor del *action-value* $q_\pi(s, a)$, si se almacenan los valores medios para cada acción en cada estado. Este tipo de estimaciones se conocen como métodos *Monte Carlo* porque implican la media sobre todas las muestras aleatorias de las recompensas actuales.

Una de las propiedades fundamentales de las funciones de valor que se usan en el aprendizaje por refuerzo y en la programación dinámica es que satisfacen relaciones de recursividad. Esta propiedad se puede aplicar de forma similar a la que se utiliza en la ecuación (2.9) pero para la *value function*:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma \, \mathbb{E}[G_{t+1} \mid S_{t+1} = s'] \right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_\pi(s')], \text{ para todo } s \in \mathcal{S} \end{aligned} \tag{2.13}$$

Esta última ecuación (2.13) se conoce como la *Ecuación de Bellman* para el valor de v_π. En ella se expresa la relación entre el valor de un estado y el de sus sucesores. La representación gráfica de las ecuaciones de Bellman se suelen interpretar mediante diagramas de *backup*. En la figura 2.5 se muestran el diagrama de *backup* para v_π y q_π. En ambos diagramas se muestran los estados como los círculos abiertos, mientras que los círculos solidos representan pares de estado-acción. El diagrama de v_π se puede interpretar de forma que empezando desde la raíz (estado s) el agente debe tomar una de las acciones posibles basándose en su política. Desde ese punto, el entorno reacciona ofreciendo un nuevo estado s' y una recompensa r dependiendo de su dinámica fijada

por p. La ecuación de Bellman realiza un promedio a lo largo de todas las posibilidades, utilizando diferentes pesos dependiendo de la probabilidad de ocurrencia. El nombre de diagramas de *backup*, proviene del hecho de calcular el valor de v_π o q_π en la dirección ascendiente o de retorno. De un modo similar se explica el diagrama correspondiente a q_π, pero teniendo en cuenta que en este caso la raíz se corresponde con la acción elegida a para el estado s, por lo que se calcula el valor para la acción-estado.

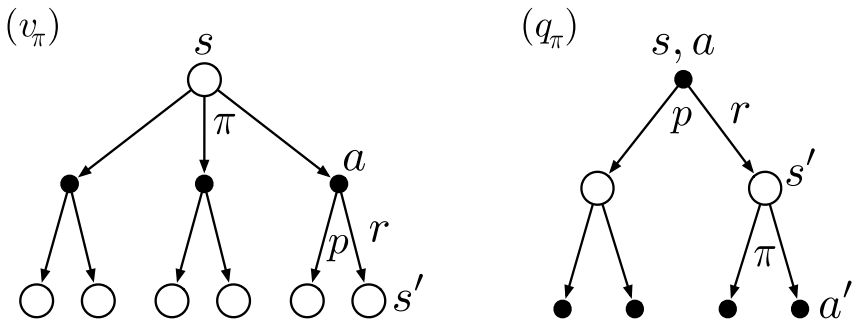

FIGURA 2.5: Diagramas de *Backup* para v_π y q_π [2]

2.3.1. Políticas óptimas y funciones de valor óptimas

La finalidad de aplicar *RL* a una tarea es conseguir el máximo número de recompensas a largo plazo. Para los *MDP* finitos, se puede definir una política óptima del siguiente modo: una política π se define mejor o igual que una política π', si su recompensa es mayor o igual que la que obtiene π' para todos los estados. En otras palabras $\pi \geq \pi'$ si y solo si $v_\pi(s) \geq v_{\pi'}(s)$ para todo estado $s \in \mathcal{S}$. Siempre hay al menos una política que es mejor o al menos igual que el resto. Esta política se conoce como política *óptima*. No obstante, puede haber más de una y se denota a toda política *óptima* como π_*. Todas estas políticas comparten la misma función de estados, conocida como la *optimal state-value function*, denotada por v_* y definida por:

$$v_*(s) \doteq \max_\pi v_\pi(s) \text{ , para todo } s \in \mathcal{S} \tag{2.14}$$

Las políticas óptimas también comparten la misma *action-value function* óptima, denotada por q_* y que se define como:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \text{ , para todo } s \in \mathcal{S} \text{ y } a \in \mathcal{A} \tag{2.15}$$

Para cada par estado-acción (s,a), la función *action-value* proporciona el valor esperado de recompensas de tomar la acción a en el estado s y a continuación seguir la política óptima. Por esto se puede escribir q_* en términos de v_* como sigue [2]:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{2.16}$$

Teniendo en cuenta que v_* es una función de valor para una política, debe satisfacer la condición de consistencia proporcionada por la ecuación de Bellman (2.13). Además, dado que se corresponde con el valor óptimo de la función de valor, la ecuación de v_* se puede escribir de forma que no haga referencia a ninguna política en especifico. Esta forma de expresar la función de valor se conoce como la *ecuación de Bellman de*

optimalidad (*Bellman optimality equation*). Intuitivamente, la ecuación de optimalidad de Bellman expresa la forma en la que el valor de un estado bajo una política óptima debe ser igual a la recompensa esperada para la mejor acción en un estado [2].

$$\begin{aligned} v_* &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.17}$$

Las dos últimas formas de expresar la ecuación (2.17) corresponden con las ecuaciones de optimalidad de Bellman para v_*. Por otro lado, la ecuación de Bellman óptima para q_* es la siguiente:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{2.18}$$

Los diagramas de *backup* para el caso de las ecuaciones óptimas de Bellman se representan en la figura 2.6. Es una representación similar a la ofrecida en la figura 2.5, pero teniendo en cuenta, que en este caso que el agente elije la acción con mayor recompensa esperada. Para denotar esta elección, se añaden los arcos sobre todas las acciones posibles. En la figura 2.6 se representa a la izquierda el diagrama correspondiente con la ecuación de Bellman para v_* (2.17), mientras que a la derecha se representa la ecuación para q_* (2.18).

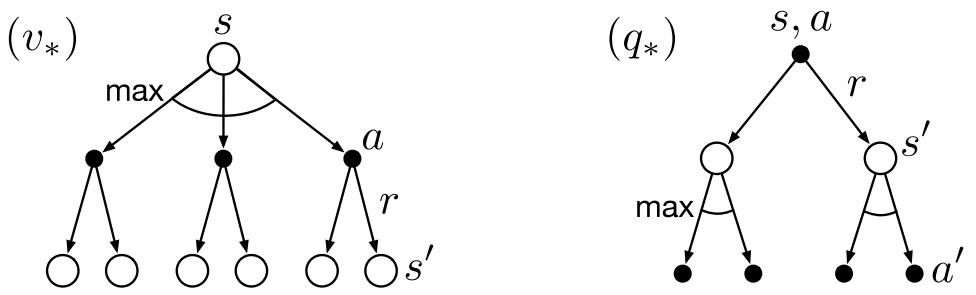

FIGURA 2.6: Diagrama de *Backup* para v_* y q_* [2]

Para un *MDP* finito, la ecuación de optimalidad de Bellman de v_π (2.13) tiene una solución única independientemente de la política. La ecuación de optimalidad Bellman se corresponde entonces con un sistema de ecuaciones, una para cada estado, por lo que si hay n estados existirán n ecuaciones con n incógnitas. Si la dinámica del entorno p es conocida, en principio, se puede resolver el sistema de ecuaciones para v_* usando uno de entre una variedad de métodos existentes para resolver ecuaciones no lineales. Del mismo modo se puede utilizar para resolver q_*.

Una vez se encuentra v_*, es relativamente sencillo determinar la política óptima. Para cada estado s, puede haber una o más de una acción en la que el máximo se obtiene de la ecuación de optimalidad de Bellman. Cualquier política que asigna probabilidad distinta de cero a estas acciones se corresponde con una política óptima. El aspecto interesante de utilizar v_* es que se utiliza para evaluar a corto plazo las consecuencias de las acciones (acciones en un paso). Por lo tanto, una política *greedy*[2] obtiene las recompensas no solo basándose en las mejores acciones actuales sino a largo plazo gracias al uso de v_*. Por otro lado, si se dispone de q_*, elegir las acciones óptimas es incluso más sencillo. Utilizando q_*, el agente no tiene que realizar un paso adelante para obtener la acción que maximiza el camino. En este caso, en cada estado s la acción óptima corresponde con la acción que maximiza $q_*(s, a)$ [2].

El hecho de conocer la función de valor de las acciones por estado o *state-action value function*, permite tomar decisiones de forma óptima incluso si no se conoce la dinámica del entorno. Este hecho resulta muy interesante en la teoría de *RL*, permite separar completamente el entorno del agente y lograr un control óptimo por parte del agente que desconoce completamente la dinámica del entorno.

El modo de obtener la función de valor del estado o el par estado-acción se puede lograr por múltiples procedimientos utilizados en *RL* [2, 4, 5]. Una vez se conoce la función de valor de cada estado o estado-acción, la política únicamente debe de seleccionar la acción que maximiza la recompensa. Este tipo de procedimientos se conocen como tabulares. Se puede considerar cada fila la representación de un estado y cada columna la representación de una de las acciones posibles en ese estado. El agente únicamente debe elegir dependiendo del estado en el que se encuentre (fila correspondiente) la acción (columna) que maximiza el valor esperado de recompensa.

Por otro lado, la reproducción de una función de valor se vuelve prohibitiva cuando el número de estados crece en exceso. En este punto, el número de ecuaciones que son necesarias para resolver el *MDP* al completo crece de tal manera que es imposible resolverlo con la potencia de computación actual. Por este motivo son ampliamente utilizados los algoritmos capaces de aproximar una función no lineal correspondiente a las n ecuaciones anteriormente comentadas. Para realizar esta tarea de aproximación se emplean en abundancia las aproximaciones por redes neuronales.

2.4. Aproximación de funciones

La aproximación de funciones se utiliza en el ámbito de *RL* cuando el espacio total de los distintos estados es arbitrariamente grande. En muchas de las tareas que se pretenden solucionar por medio del aprendizaje reforzado, el espacio de estados es demasiado grande, y como consecuencia resulta imposible el encontrar una función de valor óptima incluso disponiendo de tiempo infinito. El propósito de aproximar funciones es obtener una buena aproximación a la función de valor con los recursos computacionales actualmente disponibles.

[2] *Greedy correspondiente a un concepto de programación dinamica en la que se escojen las acciones con mayor recompensa, las "mejores" acciones o más "agradecidas".*

En la tarea desempeñada de *RL*, es posible que cada estado visitado pueda ser distinto, puede corresponder con un estado nunca visto con anterioridad. Para poder realizar decisiones sensibles en muchos de los casos, es necesario tener en cuenta el hecho de la **generalización**. Esta generalización permite poder ofrecer una respuesta a estados que no han sido visitados con anterioridad. Teniendo en cuenta los conocimientos de los que se dispone se realiza una aproximación al nuevo estado, de alguna forma que tenga sentido con alguno de los estados ya visitados con anterioridad y del que se disponga de un valor óptimo. De este modo, gracias a la experiencia con un subconjunto de datos, se puede generalizar para producir una buena aproximación a un subconjunto de datos completamente distinto al proporcionado. Para ello, se pueden combinar los métodos de aprendizaje por refuerzo con los mecanismos de generalización existentes. Estos métodos de generalización se conocen como **aproximación de funciones** o *function approximation* ya que utilizan una cierta cantidad de ejemplos para obtener una función que aproxime cualquier dato de entrada nuevo a la salida correcta. La aproximación de funciones es un caso del aprendizaje supervisado o *supervised learning* y cualquiera de sus métodos pueden ser aplicados en *RL*, a pesar de que en la práctica unos se adaptan mejor que otros.

En resumen, lo que se pretende obtener es una aproximación de las funciones de valor ya sea estado o acción-estado como se muestra a continuación:

$$\begin{aligned} \hat{v}(s, \mathbf{w}) &\approx v_\pi(s) \\ \hat{q}(s, a, \mathbf{w}) &\approx q_\pi(s, a) \end{aligned} \tag{2.19}$$

Dentro de la aproximación de funciones, se pueden determinar tres tipos de aproximaciones. Estos tipos se muestran en la figura 2.7 donde a la izquierda se representa la aproximación de la función de valor dependiente del estado. En el medio se encuentra la aproximación dependiente del estado-acción, y a la derecha la aproximación de todas las posibles funciones de estado teniendo en cuenta las posibles acciones para cada estado. En principio cualquiera de ellas es útil para encontrar la política óptima, no obstante la función más rápida a la hora de encontrar el principio de optimalidad corresponde con calcular la función de estado-acción o q_π. Si se calcula para una acción (opción más rápida) y se determina que con otras posibles acciones se puede obtener recompensa mayor, origina un calculo cíclico de optimización. Pese a que solo se necesite obtener una salida, es un proceso más costoso en cuanto a tiempo. Si por el contrario se calculan las salidas para todas las acciones posibles, únicamente se debe elegir la opción que maximiza el valor de $\hat{q}(s, a_i, w)$ para obtener el comportamiento óptimo. Este último caso representado a la derecha de la figura 2.7 es el más utilizado y el más conveniente en la mayoría de los casos.

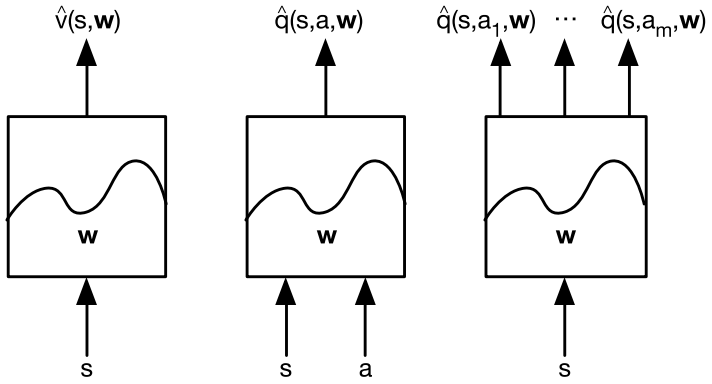

FIGURA 2.7: Tipos de aproximación de la *Value Function* [6]

2.4.1. Aproximación lineal

La aproximación de una función lineal para obtener la función de valor corresponde con el método más simple de aproximación de funciones. En este método, se aproxima la función de valor expresada como su estimación $\hat{v}(\cdot, \mathbf{w})$ donde $\mathbf{w}$[3] representa un vector de pesos lineal. Correspondiente con cada estado $s \in \mathcal{S}$ existe un vector de características o *features* que representan dicho estado, denotado por el vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^T$ con el mismo número de componentes que $\mathbf{w}$. De este modo la aproximación lineal se expresa como el producto entre $\mathbf{w}$ y $\mathbf{x}$(s):

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s) \tag{2.20}$$

El vector $\mathbf{x}$(s) también se conoce como el vector de características o *feature vector* que representa el estado s. Cada componente de $x_i(s)$ de $\mathbf{x}$(s) es el valor de una función de $x_i : \mathcal{S} \to \mathbb{R}$. Existen multitud de formas, estudios, o técnicas de obtener estas *features*. Dependiendo del tipo de tarea a realizar será necesario implementar lo que se conoce como ingeniería de *features* o realizar una extracción de *features* de modo que se pueda representar de forma óptima el estado actual. Esta representación del estado no es característica de la aproximación lineal sino que se utiliza en el resto de métodos de igual manera. Para métodos lineales, las *features* son **funciones base** debido a que forman una base lineal para el conjunto de las funciones aproximadas. La construcción de vectores de características *d-dimensionales* para la representación de los estados es lo mismo que seleccionar un conjunto de d funciones base [2]. Existen multitud de métodos para extraer las *features* necesarias para la aproximación lineal como por ejemplo *features polinómicas*, bases de Fourier, codificación gruesa, etc.

[3] En otros documentos o libros se puede encontrar este vector de pesos de distintas formas como θ, pero el significado es el mismo.

Para la aproximación lineal es muy normal el uso de *SGD* para realizar las actualizaciones de la función de aproximación. El gradiente que aproxima la *value function* con respecto a los pesos se deriva de la ecuación (2.20) como sigue:

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s) \tag{2.21}$$

Para explicar este hecho, se tiene por otro lado una función diferenciable que es función del vector de parámetros $\mathbf{w}$: $J(\mathbf{w})$. Se define por lo tanto el gradiente de dicha función como:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \cdots \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \right)^T \tag{2.22}$$

Para encontrar el mínimo local de la función $J(\mathbf{w})$ se ajustan los pesos de forma iterativa. Para ello se ajusta dicho vector de pesos $\mathbf{w}$ en la dirección negativa al gradiente (dirección de descenso). El ajuste necesario en el vector de pesos se muestra en la siguiente ecuación:

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \tag{2.23}$$

El objetivo final de la búsqueda del mínimo se determina normalmente con la minimización del error cuadrático medio (*MSE*) entre la función aproximada $\hat{v}(s, \mathbf{w})$ y la función real $v_\pi(s)$. Esta función de minimización del error cuadrático medio se puede expresar como:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, \mathbf{w})^2 \right] \tag{2.24}$$

El teorema del descenso de gradiente encuentra el mínimo local de error cuando se juntan las ecuaciones de la función de costes (2.24) y la del incremento del vector de pesos (2.23):

$$\Delta \mathbf{w} = \alpha \, \mathbb{E} \left[(v_\pi(s) - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) \right] \tag{2.25}$$

Finalmente, en el caso de descenso estocástico lo que se hace es muestrear directamente el gradiente. Esto tiene sus ventajas e inconvenientes pero se asegura de que si el número de muestras que se realizan es suficientemente grande, la convergencia al mínimo del error corresponde con la aproximación correcta de la política óptima.

$$\Delta \mathbf{w} = \alpha (v_\pi(s) - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) \tag{2.26}$$

2.4.2. Aproximación no lineal

De forma más compleja, las funciones de valor se pueden aproximar mediante funciones no lineales. Estas funciones no lineales son ampliamente utilizadas gracias a la implementación de las redes neuronales artificiales o *ANN*. Las *ANN* son redes de interconexión de unidades con ciertas propiedades similares a las de las neuronas. Estas redes neuronales tienen una amplia historia en cuanto a investigación. Los últimos avances en redes neuronales profundas facilitan la aproximación de funciones de todo tipo con una complejidad computacional permisible.

Las redes neuronales se componen principalmente de tres partes: una capa de entrada, n capas ocultas y una capa de salida. Cada "neurona" que forman las capas, corresponde con una función típicamente semi-lineal. Esto se debe a que computan la suma ponderada de la señal de entrada a dicha "neurona" y aplican dicho resultado a una función de salida no lineal conocida como *función de activación* o *activation function*. Existen distintos tipos de funciones de activación con distintos resultados esperados como la señal *sigmoidea*, función logística, etc. Del mismo modo se pueden encontrar distintos tipos de interconexiones entre las distintas capas de las redes neuronales. Con esta distinción se pueden encontrar redes neuronales *fully connected layer*, redes convolucionales, recursivas, etc. Cada tipo de red neuronal es util para un tipo de problema, teniendo que realizar una decisión inicial conociendo las características de cada una de ellas.

En el apartado 3.6 se tratará el tipo de red en mayor profundidad con la descripción necesaria para el caso utilizado. Por el momento, es suficiente con conocer que este tipo de redes neuronales son capaces de aproximar las funciones no lineales descriptivas en busca de cada política óptima.

2.5. *Deep Q Learning*

El término de *Deep Reinforcement Learning* o en español aprendizaje por refuerzo profundo hace referencia al uso de redes neuronales profundas para aplicar la teoría de aprendizaje por refuerzo. Uno de los tipos de aprendizaje por refuerzo profundo más utilizados es la implementación del algoritmo *Q-learning*. Este algoritmo trata de *aprender* el valor de la función de aproximación del valor estado-acción óptima o $q_*(s, a)$ independientemente de cual sea la política seguida. A pesar de esto, la política sigue teniendo un efecto importante en cuanto a la decisión de los estados visitados. Se describe Q como una función de estado-acción que devuelve un valor real: $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. El valor óptimo de la función ($Q^*(s, a)$) significa que si se está en el estado s y se ha tomado la acción a se obtiene el valor máximo esperado de recompensa, siempre que a partir de ese punto el comportamiento continúe siendo el óptimo. Por lo tanto, $Q^*(s, a)$ representa el cómo de bueno es para un agente elegir una acción estado en el estado s. Esto ofrece una relación directa entre la función de valor en un estado y la función Q de la siguiente forma:

$$v^*(s) = \max_a Q^*(s, a) \tag{2.27}$$

Se puede extraer la política óptima simplemente con conocer la acción que maximiza el valor esperado de recompensas para ese punto. Es interesante remarcar que el aprendizaje de la función Q o *Q-learn* tiene la propiedad de ser un algoritmo sin necesidad de modelo representativo o *model-free*. Por lo tanto, el agente descubre las recompensas de las acciones por medio de prueba y error. Los algoritmos de *Q-learning* surgen de los algoritmos de control *Temporal-Difference learning* y su actualización se defince como:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.28}$$

Para realizar un control por medio de los algoritmos de *RL*, basta con utilizar las funciones correspondientes a cada método con una actualización incremental. Esta actualización corresponde con el incremento de los pesos explicado en la ecuación (2.26) y se puede aplicar a cada uno de los algoritmos. Por ejemplo en el caso de *Q-learning* cuando $\alpha = 1$ siendo el caso más sencillo,la actualización se despeja de la ecuación (2.28) como:

$$Q(S_t, A_t) = R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) \tag{2.29}$$

La ecuación (2.29) se corresponde con la actualización llevada a cabo en los algoritmos implementados de $Q-learning$ del código del detector de anomalías. No obstante se pueden implementar distintos algoritmos de actualización incremental sustituyendo la ecuación del incremento de pesos en la ecuación del $target$ de $q_\pi(s, a)$:

$$\begin{aligned}
&\textit{MonteCarlo}: \\
&\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \\
&\text{TD(0)}: \\
&\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_t t + 1, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \\
&\textit{Forward-view} \text{ TL}(\lambda): \\
&\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})
\end{aligned} \tag{2.30}$$

Una de las formas de ver la aplicación del *Q-Learning* es en la conocida como ***DQN***. Este término queda definido la empresa deepmind en su artículo publicado en la revista Nature [7]. Este término implica la utilización de redes neuronales profundas junto con aprendizaje reforzado. La finalidad del proyecto era la de controlar diversos juegos de la consola Atari 2600 a un nivel de "superhumano". Para que los agentes desarrollados sean considerados verdaderamente inteligentes deben demostrar que desempeñan tareas "complejas" para los humanos de una forma competitiva. Hasta este punto, solo se había logrado crear programas específicos únicamente para una tarea, capaz de vencer a un humano. Este desarrollo supone un punto de inflexión, debido a que lo que se logra con *DQN* supone entrenar una maquina para poder desarrollar sus conocimientos en base a unos píxeles recogidos de la pantalla. Con esta tecnología se logró vencer a jugadores profesionales en un rango de 49 juegos distintos con el mismo algoritmo. La variedad de juegos en la que el algoritmo resultó ganador se puede ver en la figura 2.8. En la misma figura también se aprecia las mejoras que ofrece *DQN* frente a una aproximación lineal (que solo se desenvuelve mejor en *Double Dunk* y en este caso no son mejor que un humano).

Como consecuencia a este resultado, se ha utilizado la tecnología de *DQN* en una amplia variedad de tareas debido a su facilidad de desenvolverse en nuevos escenarios. Para ello *DQN* se puede entender como la suma de varias tecnologías de *RL*. El hecho de implementar *DQN* implica en teoría utilizar *Experience Replay*, *DDQN*, redes convolucionales, etc. No obstante, esto no es siempre necesario ya que cada mejora a la simple *Q-Learning* implica solucionar un problema que se les planteó para conseguir su objetivo. En todo caso, se explicará a continuación todas las tecnologías implementadas y el porqué de ellas. La compleja red neuronal implementada por *deepmind* para este artículo se muestra en la figura 2.9.

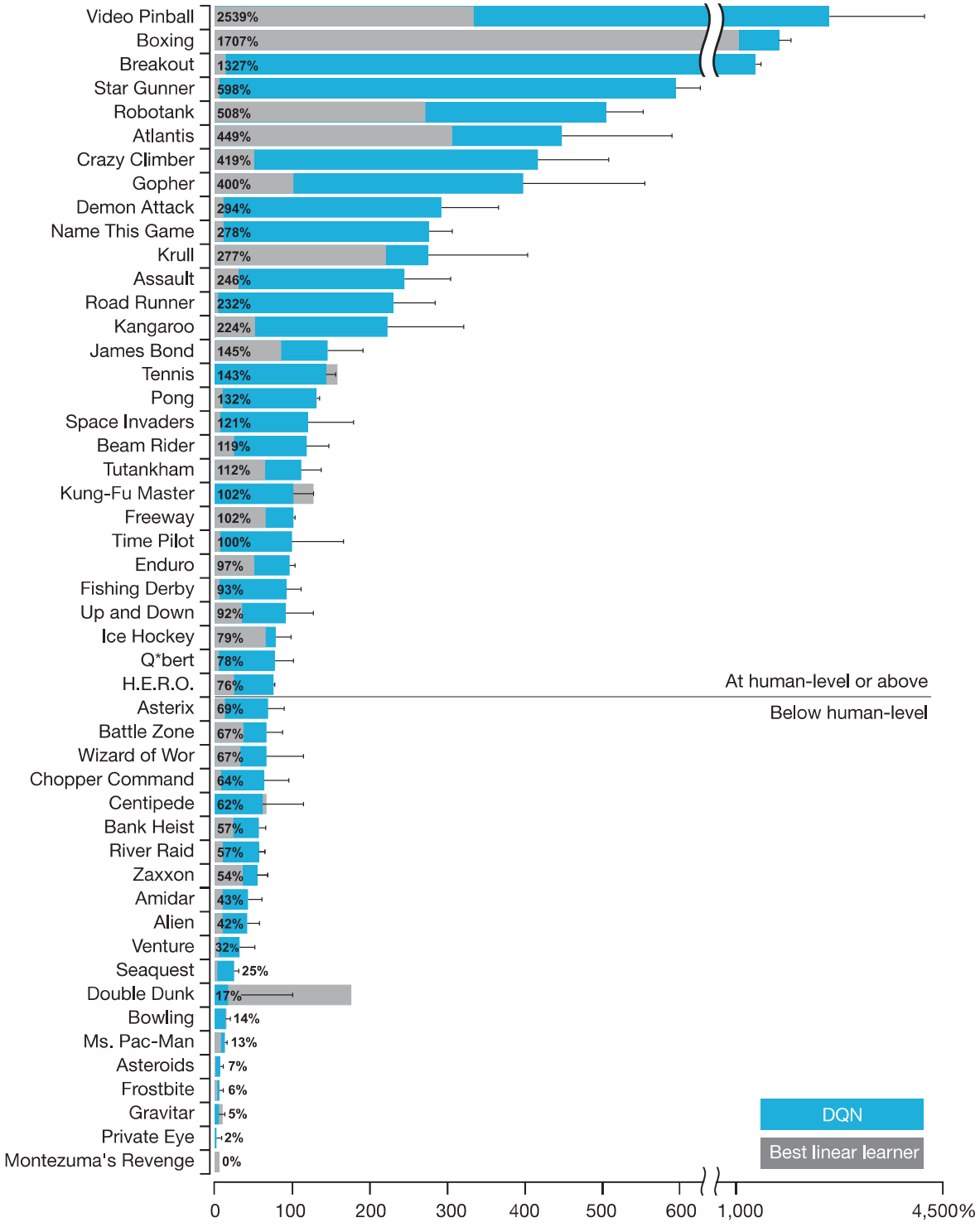

FIGURA 2.8: Resultados en *Human-level control through deep reinforcement learning* [7]

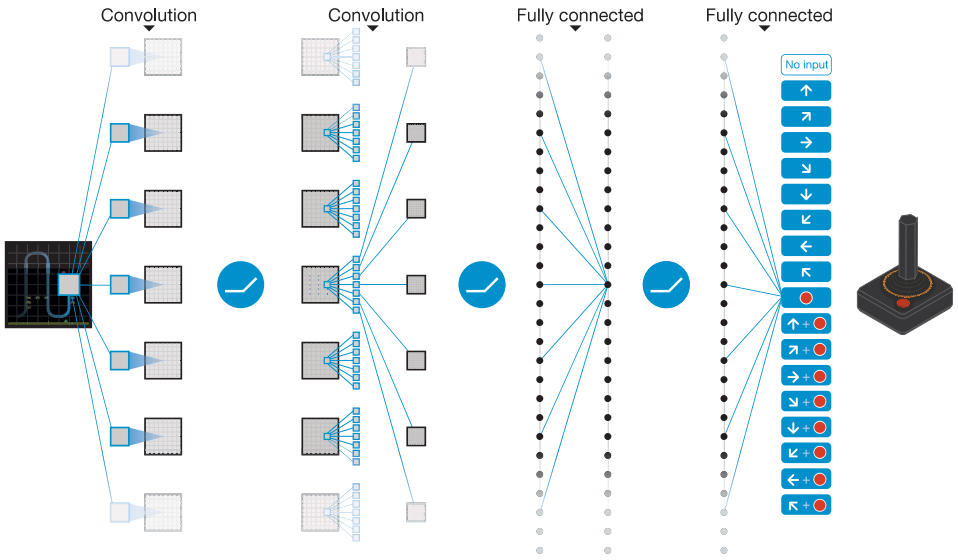

FIGURA 2.9: Arquitectura de la red neuronal *Human-level control through deep reinforcement learning* [7]

2.6. Elección de la política: *Exporation-Exploitation*

La característica más importante que distingue el *RL* de otros tipos de aprendizaje es que utiliza la información del entrenamiento que evalúa las acciones tomadas en lugar de aprender con acciones correctamente etiquetadas. Esto crea la necesidad de una exploración activa de la búsqueda del buen comportamiento. La retroalimentación ofrecida para la evaluación, indica como de buena fue la acción llevada a cavo, pero no indica si es la mejor o la peor acción posible tomada. Esto lleva al dilema de la exploración y explotación.

El dilema de la exploración y explotación de nuevas acciones supone un gran reto para los agentes de *RL*. Este dilema ha sido muy estudiado y ofrece muchas posibles opciones como soluciones al problema. Ninguna de ellas se corresponde con la solución óptima para el problema, no obstante, algunas se desenvuelven mejor que otras dependiendo del entorno. El hecho de tener que tomar decisiones óptimas es anterior a la teoría de *RL*. El propósito es siempre el mismo, intentar seleccionar las mejores acciones aprovechando o explotando el conocimiento previo sin perder la idea de otras posibles acciones que aún no se conocen y que en su objetivo final pudiesen ser mejores.

El ejemplo más estudiado para dar solución a este problema es el de "*multi-armed bandit*", más en especifico el "*k-armed bandit*". Este problema se detalla cómo k maquinas de slot, en las que elegir la acción implica jugar en dicha máquina. Cada acción o jugada en esa maquina tiene como consecuencia una recompensa o valor recibido. La

finalidad de este problema es obtener tras cierto número de repeticiones (denotado como T instantes temporales) la mayor recompensa esperada concentrando las acciones en la máquina que ofrece las mejores recompensas.

Suponiendo que se conocen las distribuciones de recompensa para cada acción aplicada $\mathcal{P}(r \mid a)$, la política óptima del entorno siempre tomará la acción con mayor esperanza de retorno $a^* = \text{máx}_{a \in \mathcal{A}} \mathbb{E}[r \mid a]$. Esto se puede interpretar de modo que la política óptima debe explorar entre todas las posibles acciones, observar sus distribuciones de recompensa y después escoger aquella que ofrece mayor recompensa esperada. En la práctica, esto es imposible ya que no se conocen las distribuciones de recompensa verdaderas, por lo que es imposible aplicar esta estrategia. Para solventar su falta, se utilizan las funciones de minimización del *regret*. El minimizar el *regret* implica del mismo modo maximizar la recompensa total. Este *regret* se puede definir como la diferencia entre seguir un política óptima frente a la política seguida o lo que es lo mismo, señala el coste (en este caso en función de la recompensa) al que estamos renunciando al no seguir una política óptima en cada instante temporal. Entonces, se puede definir el *regret* como el número de instantes temporales T por la esperanza de la recompensa teniendo en cuenta que se ha seguido la política óptima (máxima recompensa posible), menos la recompensa obtenida en cada instante de tiempo teniendo en cuenta que se ha aplicado la acción $a \in \mathcal{A}$:

$$\textit{Regret} : L_T = T\,\mathbb{E}[r \mid a^*] - \sum_t \mathbb{E}[r \mid a_t] \tag{2.31}$$

De la ecuación (2.31) se puede deducir que maximizar la recompensa obtenida $\sum_t \mathbb{E}[r \mid a_t]$ implica minimizar el *regret*. De este modo, la política que se puede alcanzar que origina un *regret* de cero corresponde con la política óptima. A continuación se presentan distintos algoritmos para la exploración-explotación.

2.6.1. *Greedy*

Los algoritmos de *Greedy* son los más sencillos para implementar la política del agente. Este tipo de política selecciona siempre la mejor opción entre todas las que dispone, por lo que la exploración no existe. Con este tipo de políticas, se encontrará un máximo de recompensas o un mínimo de *regret* pero no será absoluto, lo que implica estancarse en un mínimo local. Una vez se selecciona un mínimo local para un estado, no se podrá mejorar la recompensa esperada por lo que existe la posibilidad de que dicho mínimo sea el óptimo pero no siempre lo será.

2.6.2. *Optimistic Greedy*

Para mejorar en cierto modo el algoritmo *Greedy*, se suele iniciar la política con lo que se conoce como un valor optimista de recompensas. De este modo, al tomar como acción la primera disponible se obtiene una recompensa menor que la *optimista*. En la siguiente iteración se escoge la siguiente acción que maximiza la recompensa y del mismo modo se debería obtener una recompensa menor que la *optimista*. Esto se repite al menos una vez para cada acción disponible (siempre que se fije el valor *optimista* de forma correcta) para asegurar que se realiza una exploración de todas las acciones.

2.6.3. *Epsilon-Greedy*

Epsilon-Greedy o ϵ-*Greedy* se corresponde con una mejora significativa de los algoritmos *Greedy* anteriores. Este tipo de forma para seleccionar la acción óptima es uno de los más utilizados en *RL* por su facilidad de implementación y buenos resultados. El funcionamiento es el siguiente:

- Con una probabilidad de ϵ se escoge una acción de forma completamente aleatoria
- En el otro caso se escoge la que maximiza la recompensa esperada

$$\begin{aligned} \mathcal{P}(\epsilon) : a_t &= \text{random } a \in \mathcal{A} \\ \mathcal{P}(1-\epsilon) : a_t &= \underset{a \in \mathcal{A}}{\text{máx}} \, \mathbb{E}[r \mid a_t] \end{aligned} \tag{2.32}$$

Todos los algoritmos *Greedy* presentan un problema y es que la minimización del *regret* es lineal, lo que implica que aprenden lento. El algoritmo de *Greedy* explota demasiado la mejor solución $\mathcal{P}(a_t \neq a^*) \geq c$. Mientras que el algoritmo de ϵ-*Greedy* explora demasiado $\mathcal{P}(a_t \neq a^*) > c$. En ambos casos, la probabilidad de que la acción que se tome sea la óptima es mayor que cero.
En el caso de ϵ-*Greedy* al menos con una probabilidad $\mathcal{P}(\epsilon)$ veces se elegirá una acción aleatoria que puede ser no óptima, entonces $(k-1)$ acciones serán sub-óptimas de entre k acciones. La probabilidad de escoger una acción sub-óptima será de $(k-1)/k$ por la probabilidad de suceso de ϵ.
En el caso de *Greedy* existe la posibilidad de que $c = 0$ siempre y cuando la primera acción visitada sea la óptima. En el caso contrario de k posibles acciones $k-1$ son sub-óptimas por lo que la probabilidad de actuar de forma sub-óptima es $(k-1)/k$.
En ambos casos se puede extrapolar que la función de minimización del *regret* será mayor que un valor que crece de forma lineal con el tiempo como se muestra en la siguiente ecuación:

$$L_T \geq cT\frac{(k-1)}{k}\Delta \tag{2.33}$$

En donde Δ se corresponde a la separación entre la segunda mejor acción y la acción óptima correspondiente a la política óptima. Esta ecuación implica que existe un límite inferior, que en el caso de las políticas *Greedy*, se corresponde con una constante por el tiempo T. Razón por la que se dice que tienen un *regret* lineal.
Los estudios de minimización del *regret* llevados a cabo aseguran que existe un mínimo con tendencia logarítmica. Por este motivo se buscan otros algoritmos que tengan un comportamiento similar con el fin de obtener los mejores resultados posibles.

2.6.4. *UCB*

Lo que los algoritmos *Greedy* pretenden es identificar todos los posibles valores para las recompensas esperadas. En realidad esto no es necesario ya que lo que únicamente interesa es el conocimiento de cual es la acción óptima en cada momento. Para obtener el comportamiento óptimo es necesario eliminar aquellas acciones sub-óptimas, tan pronto como sea posible para dejar únicamente las mejores en cada caso. De este modo, cuando se elimina una acción, se elige de entre las restantes la más óptima. Este

desarrollo se conoce como *UCB* o algoritmo *Uper Confidence Bound* [8] y se procede del siguiente modo:

1. Inicializar cada posible acción $a \in \mathcal{A}$ con su valor *Greedy* y almacenar las estimaciones de las recompensas $\hat{r}_a$ y el número de veces que se aplica cada acción n_a
2. Para las primeras k veces, aplicar cada acción una vez (asegura que se prueban todas las acciones posibles)
3. Una vez empleadas k acciones, utilizar la ecuación (2.34) para cada instante temporal t

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left\{ \hat{r}_a + \sqrt{\frac{2 \log t}{n_a}} \right\} \tag{2.34}$$

Siguiendo el algoritmo descrito, se puede lograr un *regret* logarítmico $L_T \leq c \log T$. Esto implica que el error cometido a largo plazo frente a los *greedy* es menor, siempre y cuando el número de instantes temporales T sea mucho mayor que el número de acciones totales k.

2.6.5. *Epsilon-Greedy* decreciente

El algoritmo de ϵ-*greedy* decreciente o *epsilon decay* es otro método de obtener una cota inferior a la lineal. Con este método, el paso del tiempo implica la modificación de la pendiente que se va curvando de forma negativa y se consigue asintóticamente un comportamiento logarítmico. Existen distintos tipos de implementación de este decaimiento. En el caso prácticos llevado a cavo en el proyecto se ha utilizado el correspondiente a la siguiente ecuación (2.35).

$$\epsilon_t = \text{máx} \left\{ mnimo, \epsilon_0 * decay^t \right\} \tag{2.35}$$

Del mismo modo, se pueden encontrar el decaimiento de ϵ como:

$$\epsilon_t = \text{mín} \left\{ 1, \frac{c \mid A \mid}{d^2 t} \right\} \tag{2.36}$$

Siendo c una constante mayor que cero y $d = \text{mín}_{a|\Delta_a > 0} \Delta_a$

2.6.6. *Posterior sampling*

El método de *Posterior sampling* o también conocido como *Thompson sampling* se basa en explotar el conocimiento previo. Este principio se explica de forma precisa utilizando una formulación *Bayesiana*. Suponiendo que se dispone de una información a priori de la probabilidad de todas las acciones, se escoge la que sea más óptima. De este modo, las distribuciones a posteriori de las recompensas convergen a las distribuciones verdaderas por el principio de verosimilitud (*likelihood*). Este principio para el uso en *RL* se suele utilizar cuando el entorno se presenta en episodios. Al principio del episodio se muestrea el modelo del *MDP* del conocimiento posterior y el agente sigue la política óptima para el *MDP* hasta el final del episodio. Este conocimiento posterior

se actualiza al final de cada episodio en función de las acciones, estados y recompensas observadas [9].

El conocimiento a priori utilizado para empezar las suposiciones se escoge normalmente utilizando las distribuciones conjugadas o relaciones de familias conjugadas (*conjugate prior*). Estas suposiciones son muy utilizadas en la estadística bayesiana y para cada tipo de distribución de las recompensas se debe escoger la familia conjugada correspondiente. Si por ejemplo en cada paso de la iteración se utiliza una señal de recompensa binaria de forma { bien actuado : +1 | mal actuado: +0 }, la recompensa de este tipo se clasificaría como una distribución de *bernoulli*. Una vez se dispone del conocimiento a priori de la distribución de recompensas se utiliza la familia conjugada por el principio de verosimilitud que para este caso es la familia *beta* [10]. En [11] se puede encontrar un ejemplo para el caso de la distribución *beta*. Este ejemplo se muestra en la figura 2.10 donde el paso de los episodios muestra que las distribuciones se van acomodando a su valor real. En este caso la curva azul corresponderá con la acción óptima y por lo tanto aquella que más recompensa ofrecerá en el futuro.

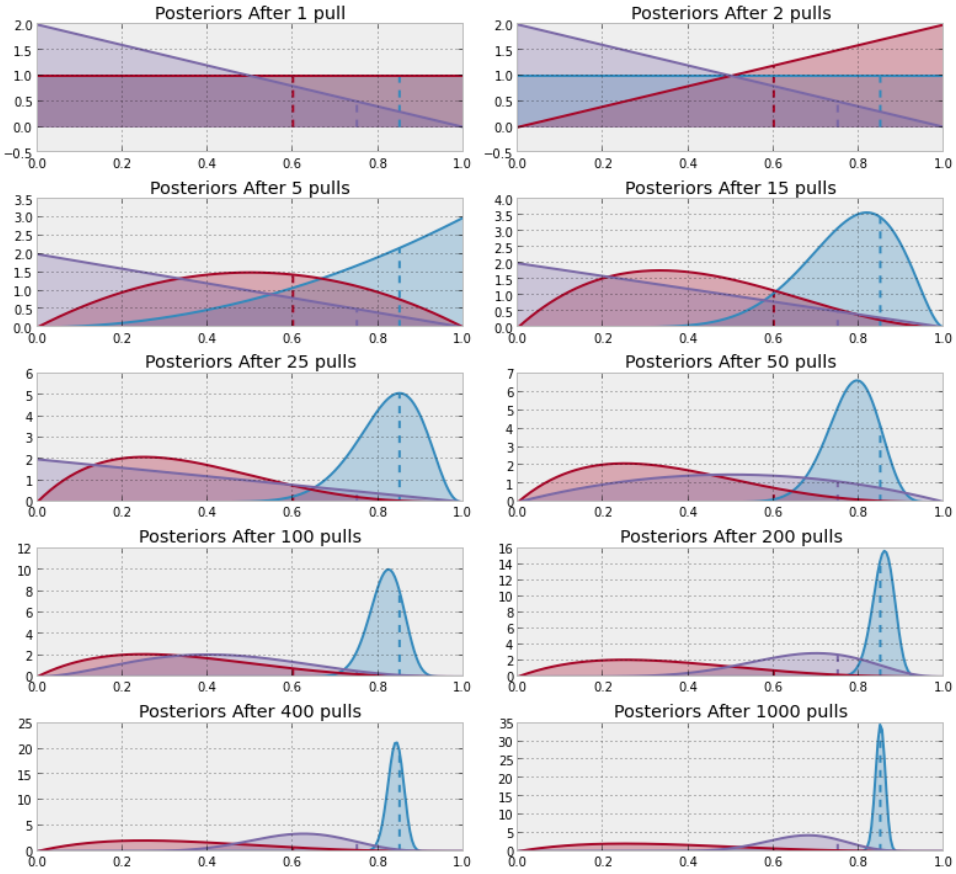

FIGURA 2.10: Evolución de la distribución beta [10]

2.7. *Policy gradient Methods*

Los algoritmos contemplados con anterioridad se han basado en el aprendizaje del valor de las acciones en cada estado (*value function*). Aprovechando éste conocimiento,

se elige la acción de mayor valor estimado siguiendo la política. Esta política no existe sin las estimaciones del valor de cada acción. La alterativa a éstos métodos, se corresponde con el aprendizaje de una política parametrizada que se encargue de seleccionar las acciones sin la necesidad de disponer de la función de valor. No obstante, la función de valor puede seguir existiendo para aprender los parámetros de la política, pero no es necesaria para la selección de las acciones.

Siendo $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ el vector de la parametrización de la política, se describe como $\pi(a \mid s, \boldsymbol{\theta}) = \mathcal{P}\{A_t = a \mid S_t = s, \boldsymbol{\theta_t} = \boldsymbol{\theta}\}$ la probabilidad de que la acción a sea utilizada en el tiempo t dado que el entorno se encuentra en el estado s en el tiempo t con el vector de parámetros $\boldsymbol{\theta}$. Si el método utilizado hace uso también de la función de valor, el valor del vector de pesos se denota como $\mathbf{w} \in \mathbb{R}^d$ como es usual en $\hat{v}(s, \mathbf{w})$ [4] [2].

Para este método, se considera el aprendizaje de los parámetros de la política como una medida de rendimiento de $J(\boldsymbol{\theta})$ con respecto a los parámetros de la política. Estos métodos tratan de maximizar el rendimiento, por lo que las actualizaciones sucesivas se aproximan directamente mediante el ascenso del gradiente en J como:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{2.37}$$

Donde $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ es una estimación estocástica donde la esperanza que aproxima el gradiente de la medida de rendimiento con respecto a sus argumentos $\boldsymbol{\theta}_t$. Todos los métodos que siguen este esquema general de la búsqueda del máximo de rendimiento, se conocen como *policy gradient methods* o métodos de gradiente de políticas.

2.7.1. *Actor-Critic*

En la figura 2.11 se puede ver una estructura de cómo utilizan los distintos métodos las aproximaciones de función. En el caso de los métodos basados en el "valor", la red neuronal se encarga de estimar la función de valor de cada estado o estado-acción. Por el contrario en los métodos basados en la estimación de la política, la red neuronal estima directamente el rendimiento de esa política en términos generales. Como situación conjunta a estos métodos se encuentra el método descrito como *Actor-Critic*. Éste, hace uso de dos estimaciones correspondientes con la función de valor y la política para obtener los siguientes beneficios:

- Mejores propiedades de convergencia
- Efectividad en espacios de acciones continuos o de altas dimensiones
- Posibilidad de aprender políticas estocásticas

[4] Siendo d' la dimensionalidad de los componentes de $\boldsymbol{\theta}$ y d la dimensionalidad de los componentes de $\mathbf{w}$.

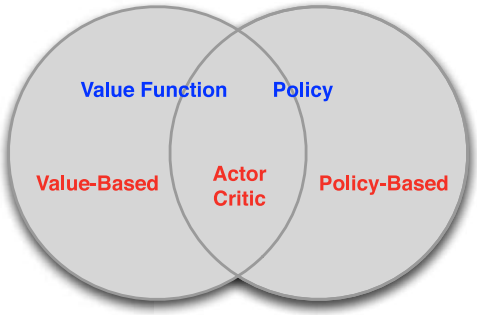

FIGURA 2.11: Estimaciones de los distintos algoritmos [12]

Existen multitud de métodos para la estimación de la política por ascenso del gradiente, pero de entre todos ellos, el beneficio de utilizar *Actor-Critic* es la posibilidad de implementar un entrenamiento online. Del mismo modo que se hace con *Q-learning* los algoritmos de *Actor-Critic* utilizan métodos completamente incrementales paso a paso y se determina la actualización como sigue [2]:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t \mid S_t, \boldsymbol{\theta}_t)}{\pi(A_t \mid S_t, \boldsymbol{\theta}_t}\\ &= \boldsymbol{\theta}_t + \alpha\left(R_{t+1} + \gamma\hat{v}(S_t, \mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t \mid S_t, \boldsymbol{\theta}_t)}{\pi(A_t \mid S_t, \boldsymbol{\theta}_t)}\\ &= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t \mid S_t, \boldsymbol{\theta}_t)}{\pi(A_t \mid S_t, \boldsymbol{\theta}_t)}\end{aligned} \tag{2.38}$$

Aprovechando la ecuación (2.38) se puede explotar la siguiente identidad:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a) &= \pi_{\boldsymbol{\theta}}(s,a)\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a)}{\pi_{\boldsymbol{\theta}}(s,a)}\\ &= \pi_{\boldsymbol{\theta}}(s,a)\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(s,a)\end{aligned} \tag{2.39}$$

Donde $\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(s,a)$ se conoce como la función de puntuación o *score function* que depende directamente de la política que se quiera estimar.

Asumiendo que la función de rendimiento que se quiere maximizar se corresponde con el total de recompensas descontadas hasta el punto, utilizando un paso en los *MDP* esta función de recompensa se puede describir como $\mathbf{r} = \mathcal{R}_{s,a}$ siendo la recompensa obtenida en el estado $s \in S$ para la acción aplicada $a \in A$. Entonces la función de rendimiento se describe como:

$$\begin{aligned}J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\mathbf{r}]\\ &= \sum_{s\in S} d(s)\sum_{a\in A}\pi_{\boldsymbol{\theta}}(s,a)\mathcal{R}_{s,a}\end{aligned} \tag{2.40}$$

Y por lo tanto el gradiente de la función de rendimiento utilizando la identidad (2.39):

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in S} d(s) \sum_{a \in A} \pi_{\boldsymbol{\theta}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) \mathbf{r} \right]\end{aligned} \tag{2.41}$$

Por último, dado que el método de *Actor-Critic* estima la función Q, se puede expresar el gradiente de la función de rendimiento como:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) Q^{\pi_{\boldsymbol{\theta}}}(s,a) \right] \tag{2.42}$$

Que se introduce en la ecuación (2.38) para obtener la actualización incremental para cada paso del algoritmo.

2.7.2. *A3C*

El método *Asynchronous Advantage Actor-Critic* o *A3C* es uno de los últimos avances en la teoría de *RL*. Ha sido lanzado por el grupo de Google **DeepMind** en 2016 por lo que es muy reciente y ya se puede decir que deja casi obsoleta la teoría de *DQN*. Entre las características destacan la simplicidad, robustez y habilidad para obtener mejores resultados que los algoritmos anteriores. Además, los algoritmos anteriores como *DQN* son capaces de resolver problemas con un espacio de observaciones amplio, pero únicamente pueden resolver un conjunto de acciones discretas y de bajas dimensiones. Muchas tareas de interés, y notablemente las relacionadas con el control físico, tienen un espacio de acciones continuo y de altas dimensiones. Mientras que *DQN* necesitaría implementar un proceso de optimización iterativo para aproximarse a una decisión continua, los algoritmos de *policy gradient* aseguran acciones tanto en el rango discreto como en el rango continuo [13]. De la descomposición de su propio nombre se detalla lo siguiente:

***Asynchronous*:** De modo contrario que *DQN*, donde un único agente representa una única red neuronal e interacciona con un único entorno, A3C utiliza múltiples representaciones del mismo con la finalidad de aprender de forma más eficiente. En A3C existe una red global, y múltiples agentes trabajadores cada uno con sus propios parámetros de la red neuronal. Cada uno de esos agentes interacciona con una copia propia del entorno a la vez que los otros agentes. La razón de que esto funcione mejor que teniendo un único agente, es que la experiencia de cada agente es independiente de la experiencia del resto. De este modo, la experiencia del conjunto es más diversa. En la figura 2.12 se muestra la representación de cómo interaccionan los distintos trabajadores con la red global.

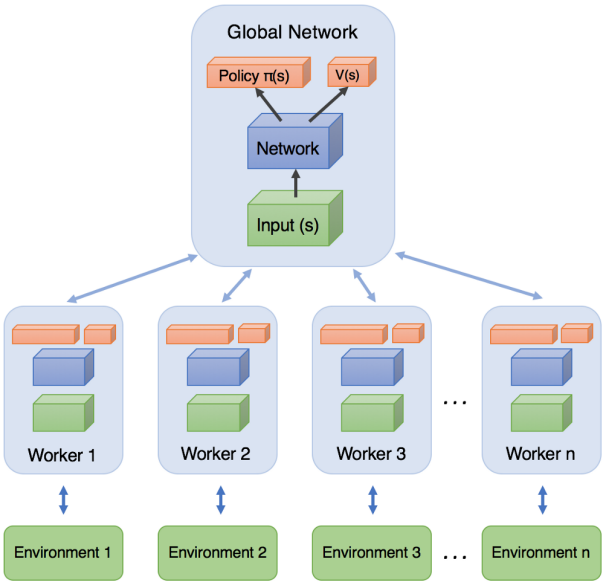

FIGURA 2.12: Diagrama de la arquitectura de A3C [14]

***Actor-Critic*:** Como se ha comentado con anterioridad, el *Actor-Critic* utiliza los beneficios tanto de los métodos basados en la funcion de valor, como los de los métodos basados en *policy gradient*. Por lo tanto, en el caso de A3C, se necesitarán dos redes neuronales, una para la función de valor que en este caso será $V(s)$ (valor de estar en un estado particular) y otra para la estimación de la política $\pi(s)$ (que puede ofrecer tanto la acción correcta como la probabilidad de tomar cada acción).

***Advantage*:** Representa la forma en la que se toman las actualizaciones y corresponde con la asignación de qué acciones son las buenas y cuales son las malas.El hecho de utilizar las estimaciones de *advantage* en vez de simplemente las recompensas descontadas, es permitir al agente determinar no solo cómo de buenas son las acciones sino en cómo de buenas se pueden convertir de forma estimada. La función de *advantage* se corresponde con la siguiente ecuación:

$$A = Q(s, a) - V(s) \tag{2.43}$$

Dado que en A3C no se determina el valor de Q directamente, se puede utilizar la función de recompensas descontadas (**R**) como estimación de $Q(s, a)$ permitiendo generar una estimación de la función *advantage* como sigue:

$$A = R - V(s) \tag{2.44}$$

Esta estimación de la función de *advantage* se puede utilizar directamente como función de aproximación lineal

$$A(s, a) = \phi(s, a)^T \mathbf{w} \tag{2.45}$$

y posteriormente utilizarla para realizar la actualización del gradiente de la política. La actualización realizada por el algoritmo puede verse como:

$$\nabla_{\boldsymbol{\theta}} \log \pi(a_t \mid s_t; \boldsymbol{\theta}') A(s_t, a_t; \boldsymbol{\theta}, \mathbf{w})^{5} \tag{2.46}$$

donde $A(s_t, a_t; \boldsymbol{\theta}, \mathbf{w})$ representa la estimación de la función de *advantage* dada por:

$$\sum_{i=0}^{k-1} \gamma r_{t+1} + \gamma^k V(s_{t+k}; \mathbf{w}) - V(s_t; \mathbf{w}) \tag{2.47}$$

donde k puede variar de estado en estado siendo el límite superior fijado por t_{max}.

En [15] se encuentra que añadir la entropía de la política ($H(s)$), mejora de la exploración por el descubrimiento de una convergencia prematura a una política determinista subóptima. El gradiente del objetivo final incluyendo la entropía y su regulación mediante el hiperparámetro β se rige por la siguiente ecuación:

$$\nabla_{\boldsymbol{\theta}'} \log \pi(a_t \mid s_t; \boldsymbol{\theta}') \left(R_t - V(s_t; \mathbf{w})\right) + \beta \nabla_{\boldsymbol{\theta}'} H\left(\pi(s_t; \boldsymbol{\theta}')\right) \tag{2.48}$$

donde $(R_t - V(s_t; \mathbf{w}))$ representa la función de *advantage* correspondiente (2.47).

Una vez se estiman los valores de las funciones, el siguiente paso es igual al resto de algoritmos, se realiza un proceso iterativo de minimización de las funciones de pérdidas. En este caso, el proceso de actualización implica modificar la estimación de la función de valor y de la política. Para ello se utiliza el descenso de gradiente para minimizar los errores con la función de valor y la recompensa asociada y por otro lado el ascenso de gradiente para maximizar la función de rendimiento de la política. Las funciones de pérdidas que se implementan en el programa son por lo tanto las siguientes:

$$\begin{aligned} L_{v(s)} &= \sum (R_t - V(s_t; \mathbf{w}))^2 \\ L_{\pi(s)} &= -\log(\pi(s_t; \boldsymbol{\theta})) A(s_t, a_t; \boldsymbol{\theta}, \mathbf{w}) - \beta H\left(\pi(s_t; \boldsymbol{\theta})\right) \end{aligned} \tag{2.49}$$

donde el signo negativo de la función de pérdidas de la política indica la dirección ascendente del gradiente y $A(s)$ se corresponde con la función de *advantage* correspondiente con la ecuación (2.44). De este modo, el **Critic** actualiza los pesos de la función de valor **w** mediante una actualización de TD(0) y el **Actor** actualiza los pesos de la función de política $\boldsymbol{\theta}$ aplicando el *policy gradient*.

[5] Nota: los parámetros $\boldsymbol{\theta}$ de la política y **w** de la función de valor, se muestran generalmente separados, pero siempre se compartirán ciertos parámetros en la práctica [15].

Capítulo 3

Implementación del detector

El propósito que se establece como finalidad del trabajo corresponde con la implementación de un detector de anomalías. Este detector como principio básico debe analizar muestras de flujos de tráfico para detectar si dicho flujo corresponde con una anomalía o se trata de tráfico normal. Para el proceso de detección se utiliza la tecnología de aprendizaje reforzado descrita en el Capítulo 2. Este detector se entrenará gracias a datos de prueba recopilados con anterioridad y que han sido ampliamente utilizados en este tipo de labores. Para ello se utilizará el dataset *KDD* y una maquina de estados que decida si se presentan anomalías o no en el estado actual.

3.1. Juego de la detección

La intención de la creación de un detector de anomalías es automatizar los procesos de detección y reacción ante estas posibles anomalías. Para ello se decide implementar dicho detector mediante el aprendizaje automático basado en aprendizaje reforzado. Este tipo de aprendizaje tiene en juego varios elementos ya comentados en el Capítulo 2. Para poder proceder con el detector, se necesita responde o contestar a una serie de preguntas:

1. Qué se va a definir como estado
2. Qué tipo de política se va a seguir
3. Qué tipo de recompensas se van a entregar
4. Cómo se obtiene el siguiente estado

En la figura 2.3 se puede apreciar las relaciones entre el agente y el entorno del problema. En este caso, el agente es el encargado de realizar las acciones. Estas acciones se podrán definir de varias formas, no obstante la más sencilla es decir si existe anomalía o no. Por otro lado, el entorno es el encargado de proporcionar la observación o representación del estado actual así como de proporcionar la señal de recompensa a la acción seleccionada. Como definición para el estado actual del sistema se utiliza cada dato proporcionado por el dataset *KDD*. Este dataset resulta de grán utilizad al estar etiquetado, de el se extraen las características o *features* y las etiquetas o *labels*. Las *features* se corresponden con los datos del flujo de una conexión y las *labels* se corresponden con el tipo de conexión realizada (normal o ataque). Tanto las *features* como las *labels* necesitan un tratamiento previo para poderlos utilizar directamente en el sistema.

3.2. KDD cup 1999

El conjunto de datos *KDD* fue utilizado para el tercer concurso internacional de herramientas para el descubrimiento de conocimientos y minería de datos (*Third International Knowledge Discovery and Data Mining Tools Competition*). Un software para detectar intrusiones en la red protege una red de ordenadores frente a usuarios no autorizados, incluyendo posibles personas con información privilegiada. La tarea de aprendizaje del detector de intrusos consiste en construir un modelo predictivo (un clasificador) capaz de distinguir entre conexiones "malas" llamadas intrusiones o ataques y "buenas" o conexiones normales. Este tipo de detectores se conoce también con el nombre de *IDS's* o Sistemas de Detección de Intrusiones. Estos detectores se han vuelto muy importantes en los últimos años debido a la dificultad técnica o económica de implementar los mismos niveles de seguridad en todos los ordenadores finales. Como principal característica, estos sistemas deben detectar anomalías y ataques en la red con el fin de mitigarlos en la medida de lo posible. Estos sistemas se suelen equipar junto con un *firewall* con la finalidad de cortar las posibles conexiones dañinas.

El dataset *KDD'99* fue generado por *DARPA* y contiene unos $4\,\mathrm{GB}$ de datos comprimidos correspondientes a tráfico originado durante 7 semanas y procesado en unos 5 millones de conexiones. El conjunto de datos de entrenamiento del *KDD* consiste en aproximadamente unas 4,9 millones de muestras con 41 *features* y se encuentran etiquetadas en normal o el tipo de ataque correspondiente. El *KDD* pretende simular una red original del *U.S Air Force* recreando el tráfico correspondiente a su *LAN* pero añadiendo distintos tipos de ataques. Entre estos tipos de ataques se pueden encontrar los siguientes:

- **Denial of Service (DoS)**: El ataque de denegación de servicio se corresponde con un ataque en el que el atacante consigue que los recursos de computación o de memoria estén demasiado ocupados o demasiado llenos para gestionar solicitudes legítimas, o deniega a los usuarios legítimos el acceso a un equipo.
- **Probing Attack**: Los ataques por sondeo son intentos de recopilar información sobre una red de ordenadores con el propósito de eludir los sistemas de seguridad con el conocimiento a priori adquirido.
- **User to Root Attack (U2R)**: Es una clase de exploit en la que el atacante comienza con el acceso a una cuenta de usuario normal en el sistema (mediante algún método de robo, fuerza bruta o ingeniería social) y es capaz de explotar alguna vulerabilidad para obtener acceso *root* al sistema.
- **Remote to Local Attack (R2L)**: Ocurre cuando un atacante sin acceso a una red tiene la capacidad de enviar paquetes a una maquina a través de la red. De esta forma explota alguna vulnerabilidad para obtener acceso local como usuario de esa máquina.

Es importante tener en cuenta que en el conjunto de datos de *test* y de entrenamiento no tienen la misma distribución de ataques, así mismo en el conjunto de *test* hay ataques específicos que no se encuentran en el conjunto de entrenamiento. Esto hace que la tarea de detección sea más realista. Algunos expertos en intrusión creen que la mayoría de los ataques novedosos son variantes de ataques conocidos y que la firma

de los ataques conocidos puede ser suficiente para detectar nuevas variantes [16]. Los conjuntos de datos contienen un total de 24 tipos de ataque de entrenamiento y 14 tipos más en los datos de prueba.

Las distintas características o *features* que se pueden encontrar en el *dataset* se pueden clasificar en tres grupos [16]:

1. ***Features* básicas** de una conexión TCP/IP. Estas se encuentran encapsuladas en todas las conexiones.

2. ***Features* de tráfico**. Esta categoría incluye las características que son calculadas con respecto al intervalo de ventana de la conexión y se subdivide en dos grupos:

   - Mismo *host*: examina sólo las conexiones que tengan el mismo *host* de destino que la conexión actual transcurridos dos segundos. Calcula estadisticas relacionadas con el comportamiento del protocolo, servicio, etc.
   - Mismo servicio: examina sólo las conexiones que tengan el mismo servicio que el actual transcurridos dos segundos.

3. ***Features* de contenido**. A diferencia de la mayoría de los ataques de *DoS* y *Probing*, los ataques *R2L* y *U2R* no tienen ningún patrón secuencial de intrusión frecuente. Esto se debe a que los ataques *DoS* y *Probing* involucran muchas conexiones a algunos *host* en un periodo de tiempo muy corto. Sin embargo, los ataques *R2L* y *U2R* están incrustados en los datos de los paquetes enviados, y normalmente, involucran sólo una conexión. Para detectar este tipo de ataques, se necesitan algunas características para poder buscar comportamientos sospechosos en la parte de datos, por ejemplo, el número de intentos de inicio de sesión fallidos.

3.2.1. Problemas

El *dataset KDD* presenta una serie de problemas que dificultan la detección de las anomalías. El problema más importante es el causado por los datos redundantes. El hecho de que los datos sean redundantes ocasiona que los algoritmos tengan un sesgo hacia los registros más frecuentes, y por lo tanto, les impida aprender los menos frecuentes. Esto implica una deficiencia mayor cuando los registros menos frecuentes se corresponden con los ataques más perjudiciales como sucede en el *KDD* (*R2L y U2R*).

3.3. Tratamiento de los datos

Para el correcto tratamiento de los datos y su posible utilización como estados es necesario realizar un proceso de ingeniería de *features*. Este proceso se realiza en todas las tareas de detección de anomalías, clasificación, análisis de datos, etc. Tener unas *features* útiles para realizar *machine learning* es necesario siempre que se disponga de datos en *bruto*. Este proceso puede ser tan complicado como se desee, pero para poder realizar una implementación rápida del algoritmo de *RL* se realizan las modificaciones básicas que permitan la utilización de estos datos como estados. De forma adicional, si se desea aumentar la detección u obtener unos datos más eficientes se pueden aplicar otras modificaciones extra. Los pasos seguidos para el tratamiento han sido los siguientes:

3.3.1. Descarga del dataset

El primer paso para obtener los datos ha sido encontrar el dataset de una fuente fiable de Internet. Existen multitud de recursos con este dataset preprocesado, pero la idea es modificarlo desde el principio. El dataset utilizado ha sido el *UCI-KDD* ofrecido por la universidad de *California*, *Irvine* [11]. Este dataset ofrece los datos separados en varios ficheros, los más importantes son:

- `kddcup.data.gz`: En este fichero, se encuentra comprimido el dataset de entrenamiento.
- `corrected.gz`: En este fichero, se encuentran comprimidas las muestras de test. Han sido etiquetadas de forma correcta cada uno de los ataques distintos debido a que en el dataset original el conjunto de test no dispone de etiquetas.
- Enlace de descarga

3.3.2. Primer análisis

Para poder procesar los datos se necesita conocer su composición. Se abre el fichero descargado y se examina la estructura y datos que lo componen. Las primeras lineas del fichero "corrected" se pueden ver en la figura 3.1 y se corresponden con un formato *CSV* que separa todas las características con comas y cada registro en una nueva línea.

```
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.00,0.00,0.00,0.00,0.00,0.00,normal.
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.00,0.00,0.00,0.00,0.00,0.00,normal.
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.00,0.00,0.00,0.00,0.00,0.00,normal.
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.00,0.00,0.00,0.00,0.00,0.00,snmpgetattack.
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,254,1.00,0.01,0.01,0.00,0.00,0.00,0.00,0.00,snmpgetattack.
0,udp,private,SF,105,146,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.01,0.00,0.00,0.00,0.00,0.00,snmpgetattack.
```

FIGURA 3.1: Primeras lineas del fichero `corrected`

3.3.3. Creación de una lista de *features*

Para el correcto tratamiento de los datos, resulta necesario disponer de una lista que posea los nombres de las *features* que han sido extraídas para su procesado individual. Para ello se crea una lista como la que se muestra en el código 3.1. Esta lista se basa en la definición de las *features* propuestas por la descripción de la tarea. Se encuentra esta lista además del tipo de dato que representa en el archivo `kddcup.names`. En este archivo también se encuentra en la primera linea los nombres de los tipos de ataques, que se detallarán a continuación.

La descripción de las *features* más relevantes se presenta en las tablas de a continuación. La tabla 3.1 presenta las características básicas de las conexiones *TCP*, la tabla 3.2 corresponde con las características de contenido de una conexión y la tabla 3.3 corresponden con las características de tráfico calculadas con la ventana *TCP*. Estas tres tablas representan la división en grupos de las características explicada en el apartado 3.2. En estas tablas también se presenta el tipo de dato con el que se corresponde. No se procesará del mismo modo los datos continuos que los datos discretos.

```
col_names = ["duration","protocol_type","service","flag",
    "src_bytes","dst_bytes","land","wrong_fragment",
    "urgent","hot","num_failed_logins","logged_in",
    "num_compromised","root_shell","su_attempted",
    "num_root","num_file_creations","num_shells",
    "num_access_files","num_outbound_cmds","is_host_login",
    "is_guest_login","count","srv_count","serror_rate",
    "srv_serror_rate","rerror_rate","srv_rerror_rate",
    "same_srv_rate","diff_srv_rate","srv_diff_host_rate",
    "dst_host_count","dst_host_srv_count",
    "dst_host_same_srv_rate","dst_host_diff_srv_rate",
    "dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate","dst_host_serror_rate",
    "dst_host_srv_serror_rate","dst_host_rerror_rate",
    "dst_host_srv_rerror_rate","labels","dificulty"]
```

CÓDIGO 3.1: Definición de las columnas

Nombre del Feature	**Descripción**	**Tipo**
duration	Longitud (en segundos) de la conexión	Continuo
protocol_type	Tipo de protocolo, ej: tcp,udp, etc	Discreto
service	Servicio de red en el destino, ej: http,telnet,etc	Discreto
src_bytes	Número de datos en bytes de la fuente a destino	Continuo
dst_bytes	Número de datos en bytes del destino a fuente	Continuo
flag	Normal o estado de error de la conexión	Discreto
land	1 si la conexión proviene o va hacia el mismo puerto/host, 0 el resto	Discreto
wrong_fragment	Número de fragmentos erróneos	Continuo
urgente	Número de paquetes urgentes	Continuo

TABLA 3.1: *Features* básicas de las conexiones TCP

Nombre del Feature	**Descripción**	**Tipo**
hot	Número de conexiones “calientes”	Continuo
num_failed_logins	Número de intentos de inicio de sesión fallidos	Continuo
logged_in	1 si se ha iniciado sesión de forma correcta 0 en el caso contrario	Discreto
num_compromised	Número de condiciones de “comprometido”	Continuo
root_shell	1 si se ha obtenido el root shell 0 en el caso contrario	Discreto
su_attempted	1 si el comando “su root” ha sido intentado 0 en el caso contrario	Discreto
num_root	Número de accesos al “root”	continuo
num_file_creations	Número de operaciones de creación de archivos	Continuo
num_shells	Número de “shell prompts”	Continuo

num_access_files	Número de operaciones en los archivos de control de acceso	Continuo
num_outbound_cmds	Número de comandos salientes en una sesión ftp	Continuo
is_host_login	1 si el acceso pertenece a una lista de "hosts" 0 en el caso contrario	Discreto
is_guest_login	1 si el acceso es de "invitado" 0 en el caso contrario	Discreto

TABLA 3.2: *Features* de contenido dentro de una conexión

Nombre del Feature	**Descripción**	**Tipo**
count	Numero de conexiones al mismo host que la conexión actual en los últimos dos segundos *Nota: Las siguientes features se refieren a las conexiones de un mismo host*	Continuo
serror_rate	% de conexiones que tienen errores "SYN"	Continuo
rerror_rate	% de conexiones que tienen errores "REJ"	Continuo
same_srv_rate	% de conexiones al mismo servicio	Continuo
diff_srv_rate	% de conexiones a diferentes servicios	Continuo
srv_count	Número de conexiones al mismo servicio que la conexión actual en los últimos dos segundos *Nota: Las siguientes features se refieren a las conexiones de un mismo servicio*	Continuo
srv_serror_rate	% de las conexiones que tienen errores "SYN"	Continuo
srv_rerror_rate	% de las conexiones que tienen errores"REJ"	Continuo
srv_diff_host_rate	% de las conexiones a diferentes host	Continuo

TABLA 3.3: *Features* de tráfico calculadas utilizando una ventana de 2 segundos

3.3.4. Adecuación de las *features*

Las *features* ofrecidas por el dataset se encuentran en el estado original en el que fueron adquiridas. Estos datos han de ser tratados ya que a una red neuronal no se pueden introducir datos en formato categórico junto con datos continuos y datos booleanos. Para unificar la totalidad de las *features*, se han procesado los datos de la siguiente manera:

- Los datos en formato **categórico**(discreto múltiple), se han convertido utilizando codificación *one-hot encoding*. Esto quiere decir que un *feature* en formato categórico se convierte en nuevas *features*. Cada *feature* dispone de un cierto número de categorías, entonces se crea una nueva *feature* para cada categoría. Una vez se dispone de estas nuevas columnas se etiqueta de forma *booleana* el registro con la *feature* a la que pertenecía anotando ese registro con un "1" a la *feature* correspondiente y con un "0" al resto.
  A este tipo de clase pertenecen: "protocol_type", "service", "flag" y "labels".

Para estimar previamente el número de nuevas features que van a aparecer se puede realizar un pequeño estudio utilizando comandos de *pandas*. Utilizando el código de a continuación se puede observar que se presentan tres tipos de protocolos distintos, esto quiere decir que tras aplicar la codificación *one-hot* para el *flag* se convertirá en once columnas distintas que se pueden ver en el código. En la tabla 3.4.

```
>>>import pandas as pd
# col_names cargado previamente.
>>>dataset = pd.read_csv('kddcup.data',sep=',',names=col_names)
>>>dataset['flag'].unique()
array(['SF', 'S2', 'S1', 'S3', 'OTH', 'REJ', 'RSTO', 'S0', 'RSTR',
       'RSTOS0', 'SH'], dtype=object)
```

Antes de one-hot											
Dato n°	**flag**										
1	SF										
2	S2										
3	SF										
4	S1										
Después de one-hot											
Dato n°	**SF**	**S2**	**S1**	**S3**	**OTH**	**REJ**	**RSTO**	**S0**	**RSTR**	**RSTOS0**	**SH**
1	1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0

TABLA 3.4: Ejemplo one-hot

- El otro tipo de datos que se encuentra en el *datset* corresponde con los datos en formato **continuo**. Este tipo de formato no es un problema a la hora de introducirlo en una red neuronal. No obstante, la amplia variedad de rangos entre los datos ocasiona un oscurecimiento de aquellos que tienen valores más pequeños. Para evitar que únicamente tengan valor útil en la red los que tienen valores más altos, se normalizan todas las *features* continuas. Con esto se consigue que todos los valores de todas las features se presenten entre "0" y "1" ó si se desea entre "-1" y "+1".

Es importante tener en cuenta que los datos que se presentan en formato binario de "1" y "0" no necesitan ser tratados, no obstante, se pueden tratar como continuos ya que la normalización si se realiza en formato "0" y "1" no va a afectar al valor.

3.3.5. Crear un mapa de ataques

Resulta muy útil disponer de un mapa de ataques para poder asignar cada ataque a su correspondiente tipo. Este mapa de ataques no se usa en la primera versión del detector ya que únicamente detecta si hay anomalía o no. Por otro, lado sí que se utiliza mucho en las siguientes versiones donde se pretende detectar el ataque o el tipo de ataque realizado.

El mapa de los ataques posibles con su respectivo tipo se muestra en la tabla 3.5 y su implementación en *python* es tan simple como asignar a un diccionario con la clave el nombre y el tipo como valor.

Nombre	Tipo
normal	normal
back	DoS
land	DoS
neptune	DoS
pod	DoS
smurf	DoS
teardrop	DoS
mailbomb	DoS
apache2	DoS
processtable	DoS
upstorm	DoS
ipsweep	Probe
nmap	Probe
portsweep	Probe
satan	Probe
mscan	Probe
saint	Probe

Nombre	Tipo
ftp_write	R2L
guess_passwd	R2L
imap	R2L
multihop	R2L
phf	R2L
spy	R2L
warezclient	R2L
warezmaster	R2L
sendmail	R2L
named	R2L
snmpgetattack	R2L
snmpguess	R2L
xlock	R2L
xsnoop	R2L
worm	R2L
buffer_overflow	U2R
loadmodule	U2R
perl	U2R
rootkit	U2R
httptunnel	U2R
ps	U2R
sqlattack	U2R
xterm	U2R

TABLA 3.5: Tabla de ataques con su tipo

3.3.6. Ajustar las etiquetas

En este paso se procede a formatear las etiquetas o *labels* en el formato deseado. En la primera versión del detector únicamente se desea detectar si existe una anomalía o no, por lo que las etiquetas serán de valor "1" si hay ataque y "0" si no lo hay. Una vez formateadas se utilizan directamente para asignar las recompensas.

3.3.7. Guardar los datos

Una vez se procesan los datos, estos se guardan en otro archivo *CSV* para que la próxima vez que se utilicen los datos ya estén procesados. Este proceso de guardado es uno de los más lentos del programa, pero si se va a ejecutar muchas veces merece la pena. Una vez están guardados los datos en un archivo, las siguientes veces que se ejecute el código simplemente se cargará el archivo en RAM.

Es interesante realizar un mezclado de los datos antes de ser guardados. Este mezclado o *shuffle* evita posibles periodicidades o patrones ocasionados en el proceso de captura. Con este mezclado se pretende aprender a detectar utilizando las características de cada flujo y no mediante su patrón de captura.

3.4. Creación del entorno

Para que funcione la detección basada en *RL*, el entorno debe de estar correctamente definido. Este entorno se define en el capítulo 2 y tal y como se aprecia en la figura 2.3 se encarga de recibir la acción propuesta por el agente. En función de esa acción actualiza los "estados" del entorno (observaciones para el agente) y genera las señales de recompensa. Por esto mismo, el entorno debe disponer de dos acciones fundamentales:

- **reset**: Encargada de situar al entorno en la posición inicial de una tarea. En este caso provee un estado inicial de forma aleatoria. También se encarga de borrar las recompensas anteriores.
- **act/step**: La función de *act*, o también se puede encontrar como *step* por su amplia apariencia en el entorno *Open AI GYM* [17]. Se encarga de dar un paso en la dirección que le implica la acción solicitada por el agente. Esta función debe devolver la siguiente observación del entorno o siguiente estado así como la recompensa que se le asigna a la acción seleccionada por el agente. También se devuelve una señal más (*done*), encargada en finalizar la tarea episódica o de entrar en un estado infinito o de absorción.

Por otro lado, los entornos suelen tener funciones de ayuda para cada realización dentro de las mismas. Se pueden implementar las siguientes:

- ***_isdone()***: Esta función indica si se ha alcanzado un estado final o de absorción.
- ***_update_state()***: Obtiene el siguiente estado teniendo en cuenta las acciones enviadas.
- ***_get_rewards()***: Se encarga de calcular las recompensas que se van a devolver.

La implementación de funciones de este tipo estructuran muy bien el entorno pudiendo modificar cualquiera de sus partes sin modificar el resto. Es posible que en un cierto punto se desee modificar el cómo se calculan los nuevos estados o las recompensas y de este modo no se altera el funcionamiento del resto. Por otro lado, se puede condensar todas estas funciones en una única o una mezcla de ellas.

Por parte del entorno existen ciertas variables que deben ser contempladas. Estas pueden estar definidas dentro del propio entorno o ser calculadas fuera con los datos que este proporciona. La primera es el tamaño de la observación del entorno o la cantidad de estados posibles que se pueden dar. Esto depende directamente de la codificación del estado, por ejemplo si se codifica con *one-hot* cada estado tendría un uno en la posición correspondiente por lo que para N estados se dispondría de una observación de tamaño N. Si se codifica de forma más eficiente se tendría una observación menor para los distintos estados. En el caso del detector de anomalías, el estado se define como las *features* proporcionadas. Esto implica que la observación del estado se

corresponda directamente con el tamaño de las *features*. Cada registro con valores distintos se corresponderá con un estado distinto. Este valor es muy importante ya que fija el tamaño de la red neuronal de entrada.

Como segundo valor importante a tratar en el entorno está el número de acciones permitidas disponibles. Dependiendo del análisis que se quiera llevar a cabo, este valor corresponderá con el número de *labels* que se hayan definido. Este valor es importante también ya que se corresponde con la salida de la red neuronal.

3.5. Creación del agente

El agente es el segundo gran componente de todo sistema de *RL*. Este agente se encarga de la toma de decisiones y sobre él recae todo el proceso de aprendizaje. Para la toma de decisiones el agente se basa en una política y debe de ser capaz de aprender y actuar de las señales devueltas por el entorno.

3.5.1. Política

Para la correcta actuación del agente se debe crear una política que permita tomar decisiones de forma que maximice la recompensa total a largo plazo. Se pueden encontrar distintas implementaciones de políticas en el capítulo 2.6.

En general, las políticas de *Q-learning* se basan en una red neuronal que aproxima la *value function*. Con esta red neuronal por un proceso de exploración explotación se escogen las acciones que maximizan el *value function*. El valor de este *value function* implica el valor medio esperado desde el punto hacia el futuro. Por lo tanto, si se obtiene el valor "aproximado" de esta función y se actúa de forma óptima, se obtendrá la máxima recompensa posible.

3.5.2. Actualización del modelo

La actualización del modelo de red neuronal de aproximación de la *value function* es otra de de las tareas del agente. El agente se encarga de recojer los valores arrojados por el entorno sobre las anteriores recompensas ante las anteriores acciones. En función de estos valores "positivos" o "negativos" debe actualizar la aproximación de la función, similar a un ajuste supervisado. Sin embargo, a diferencia de un ajuste supervisado, el agente ajusta el modelo en función de las recompensas para obtener el valor de los estados y no para ajustar el valor de la salida. La actualización de la red neuronal implica actualizar el valor de Q con la ecuación de actualización 2.29. Esto se logra con el siguiente fragmento de código:

```
# Predicción de la función de valor siguiente
Q_prime = model_network.predict(next_states)
# Mejor siguiente acción (greedy)
a_prime = np.argmax(Q_prime,axis=1)
# Guardo las posiciones por si se actualiza en batch
sx = np.arange(len(a_prime))
# Cálculo de la nueva función Q:
#   Q(s,a) = r' + gamma*max Q(s',a')
targets = reward + gamma * Q[sx,indx]
Q[sx,a] = targets
```

```
11
12 # Ajusto la función Q para el valor de los estados
13 loss += model.train_on_batch(states,Q)
```

CÓDIGO 3.2: Actualización *Q-learning*

3.6. Red neuronal de aproximación

Como se introduce en el capítulo 2.4.2, se puede obtener una aproximación no lineal de la función que representa la función de valor o *value function*. Una vez se dispone de esta función de aproximación, obtener la acción óptima se corresponde con tomar el valor para el cual se obtiene mayor recompensa futura. Este valor se corresponde directamente con la posición en la que la función tenga un valor más alto.

La estructura de la implementación de la red neuronal de aproximación para la primera versión realizada se muestra en la figura 3.3 y se corresponde con la siguiente descripción:

- Etapa de entrada con 123 unidades asociadas a las *features* del estado
- Una capa oculta *fully connected* de 100 unidades de profundidad
- Capa de salida con dos unidades correspondientes con las acciones posibles (amenaza/normal)

Esta red neuronal utiliza una función de activación de rectificación "ReLU" como la de la figura 3.2. Compila el descenso de gradiente estocástico con un parámetro de tasa de aprendizaje de $lr = 0{,}2$ y un ajuste por error cuadrático medio.
Esta implementación se realiza mediante el modelo *Sequential* de *Keras* con el siguiente fragmento de código:

```
1 # Network arquitecture
2 model = Sequential()
3 model.add(Dense(hidden_size, input_shape=(env.state_shape
      [1]-1,),
4                 batch_size=batch_size, activation='relu'))
5 model.add(Dense(hidden_size, activation='relu'))
6 model.add(Dense(num_actions))
7 model.compile(sgd(lr=.2), "mse")
```

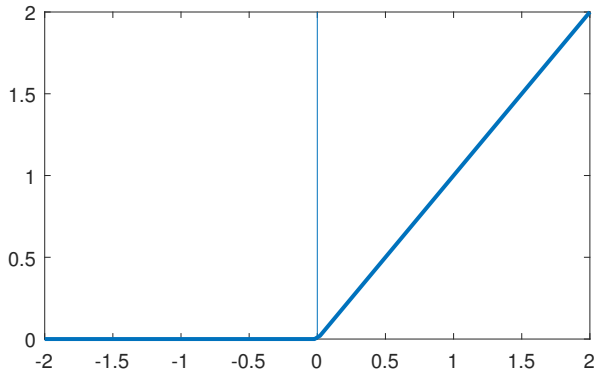

FIGURA 3.2: Función de activación rectificadora ReLU

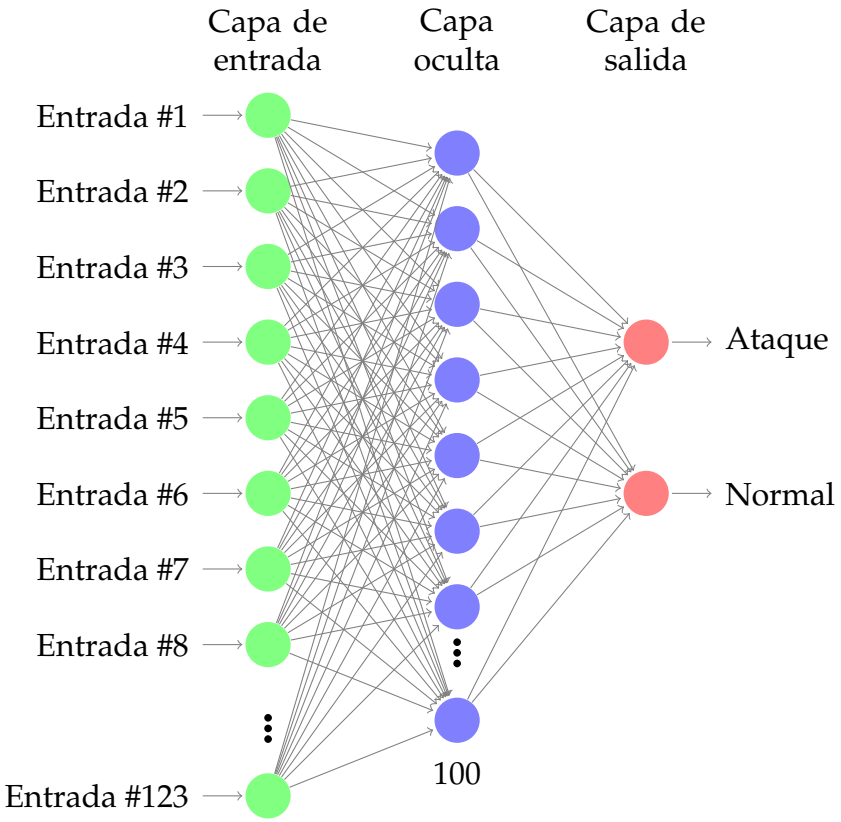

FIGURA 3.3: Estructura de la red neuronal básica

La estructura de la red neuronal con una capa oculta asegura la posibilidad de la existencia de una función de aproximación que presente no linealidades.

3.7. Detectores mejorados

A lo largo del capítulo se ha descrito el funcionamiento de un detector de anomalías simple. Este detector se encarga de ofrecer una salida binaria en la forma de "normal" o "ataque". Este tipo de detectores están muy bien para poder determinar

si se ha producido un ataque. Por otro lado, a la hora de reaccionar ante los posibles ataques resulta determinante el conocimiento del ataque que se ha llevado a cabo con la finalidad de responder de forma selecta ante él. Por este motivo, se implementan las siguientes mejoras.

3.7.1. Detección del ataque

La primera implementación que mejora el caso simple es un detector del tipo de ataque. En este caso, la salida del detector será una codificación de todos los ataques de modo que se decidirá cual ha sido el enviado. Para la implementación de este detector se han realizado las siguientes modificaciones.

Modificación en las *labels*

Con el objetivo de realizar un entrenamiento "supervisado" necesario para el algoritmo, es obligatorio realizar una modificación en el punto del tratamiento de los datos. La modificación realizada se corresponde con el apartado 3.3.6 de este capítulo. Para la detección del tipo de ataque lo que se realiza es una codificación *one-hot*, pero en este caso de cada uno de los tipos de ataques mostrados en la tabla 3.5.

Modificación de las recompensas

La forma de asignar las recompensas es muy similar al caso anterior, la única diferencia es que en este caso el número de etiquetas es mayor y, por lo tanto, la dificultad también.

Modificación de la red neuronal

La red neuronal en este caso tiene que ser capaz de ofrecer una representación del espacio de salidas correspondiente a los ataques. La forma más sencilla es mediante una codificación *one-hot* de modo que cada salida se corresponda con un ataque. El algoritmo de *RL* será por lo tanto el encargado de escoger el tipo de ataque que se ha cursado teniendo en cuenta su política.

Para la estructura de esta red, además de modificarse la salida, se ha incrementado el número de capas ocultas. Esto no siempre es necesario, pero se dispone de un dataset suficientemente elevado como para que el ajuste de pesos se realiza de forma correcta. Por lo tanto, se ha declarado el número de capas ocultas como una variable. En la figura 3.4 se muestra la estructura implementada con tres capas ocultas por ejemplo.

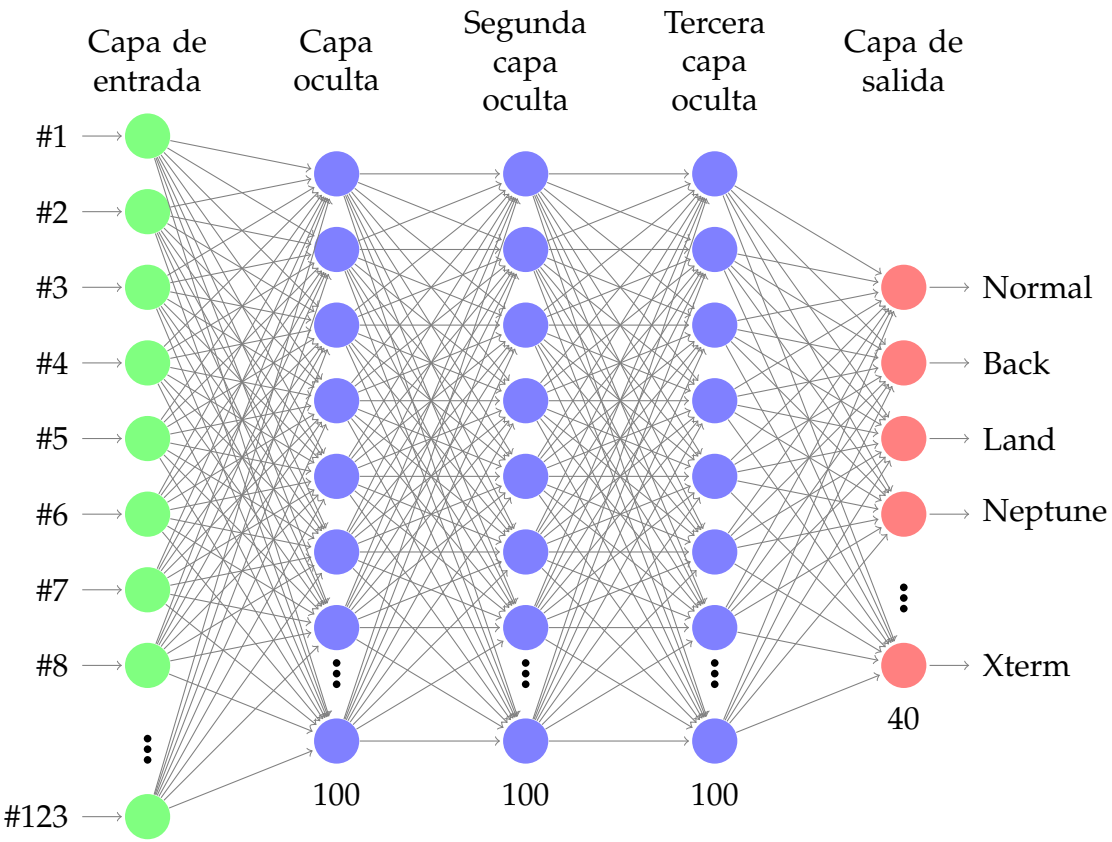

FIGURA 3.4: Estructura de la red neuronal para detectar el ataque

3.7.2. Detección del tipo de ataque

El segundo tipo de detector implementado que mejora el simple se corresponde con uno que detecta únicamente el tipo de ataque. Resulta muy interesante obtener el ataque en específico que se ha enviado, no obstante, el elevado número de distintos ataques que se presentan eleva demasiado la dificultad de detección. Si por el contrario se desea detectar únicamente el tipo de ataque enviado, el número de etiquetas se reducen a cinco. Una para el tipo normal y otras cuatro para los distintos tipos de ataques que existen. Este tipo de detector es mucho menos exigente y se obtienen tasas de detección mucho mejores. Para obtener este detector se realizan modificaciones similares a las anteriores.

Modificación en las *labels*

En este caso, el número de etiquetas son cinco, cada etiqueta se corresponde con el tipo de ataque seleccionado. Esta comparación o mapeo del tipo de ataque se realiza de forma inmediata gracias al diccionario python creado con cada ataque. Una vez se tiene la etiqueta del ataque, se almacena en el archivo de datos modificados con su tipo correspondiente. En la tabla 3.5 se muestran todos los ataques con su tipo correspondiente.

Modificación de las recompensas

Las recompensas se corresponden con un vector de cinco tipos por lo que la dificultad crece respecto al tipo simple y disminuye respecto al completo. En detalle las recompensas son de "+1" si el *label* del ataque coincide con el elegido por el algoritmo y "+0" en el caso de que no coincidan.

Modificación de la red neuronal

La red neuronal para esta situación mantiene el número de entradas como features, mantiene también el número de capas ocultas presentes y se modifica el número de salidas, en este caso cinco. El esquema de esta red neuronal se puede apreciar en la figura 3.5

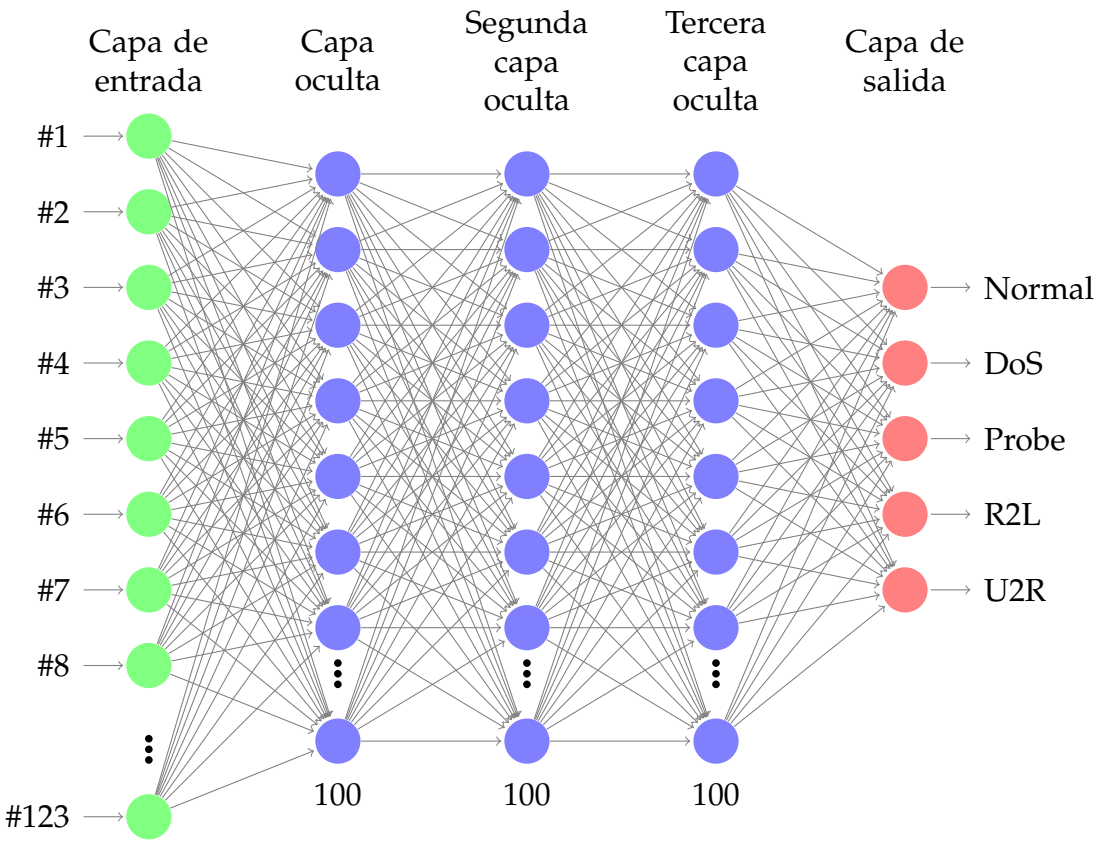

FIGURA 3.5: Estructura de la red neuronal para detectar el tipo

Capítulo 4

Resultados

A lo largo de este capítulo se muestran los resultados obtenidos de los detectores descritos en el capítulo 3. La calidad de estos detectores es completamente mejorable ya que son las primeras versiones de los algoritmos. No obstante sirve para apreciar el proceso de aprendizaje y la calidad de los algoritmos de *RL*.

4.1. Detector simple

Como primer detector analizado se tiene el caso simple. En el cual se detecta únicamente la presencia de anomalía o comportamiento normal. En la figura 4.1 se muestra el proceso de aprendizaje de la red neuronal. El sub-gráfico superior representa el total de recompensa por cada episodio y inferior se corresponde con el coste o pérdidas que trata de minimizar el algoritmo (error cuadrático medio). En este caso, cada episodio consta de un total de 1000 muestras y la recompensa por acertar es de +1. Como se aprecia en la figura 4.1 el total de recompensas se acerca mucho a los 1000 por lo que la tasa de aciertos es muy elevada. De forma opuesta a lo que puede parecer, este hecho no implica que el detector sea muy bueno. Un detector óptimo debe **generalizar** y adaptarse a cualquier conjunto de datos. El problema al que se enfrenta este detector no es culpa del propio detector sino que es culpa del conjunto de datos. Como se cita en el apartado 3.2.1, el conjunto de datos presenta muestras redundantes originando un sesgo de la detección. De todos modos, la gráfica de aprendizaje ofrece una visualización de la reducción progresiva de la función de coste y también permite apreciar la alta varianza de estos algoritmos de aprendizaje.

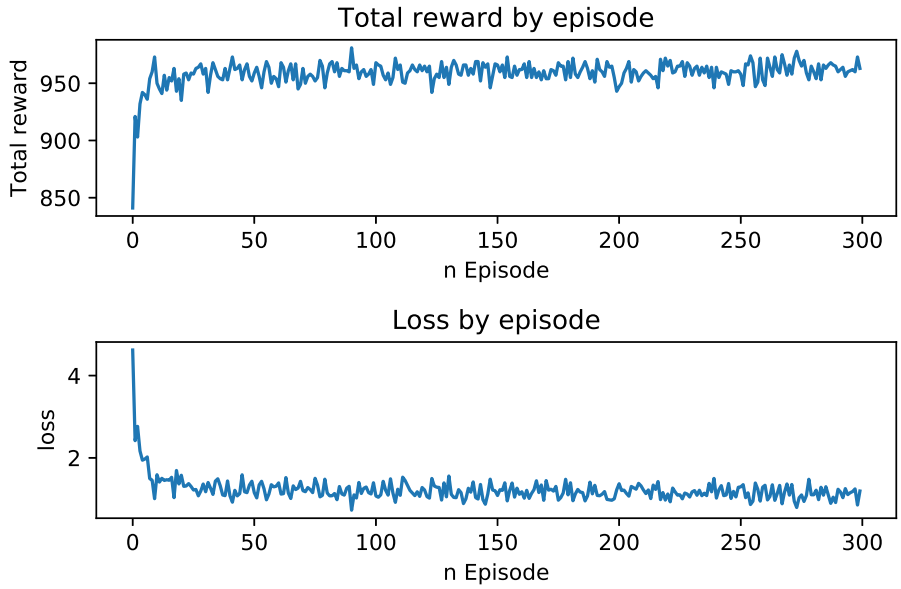

FIGURA 4.1: Fase de entrenamiento del detector simple

El ajuste de los hiperparámetros en los algoritmos de aprendizaje automático suponen uno de los procesos más importantes y tediosos para el investigador. En este caso, como el *dataset* tiene ciertos problemas, para acelerar el proceso únicamente se ajustan los necesarios para asegurar una rápida y óptima convergencia. Para lograr este objetivo, se realizan varias pruebas con distintos números de neuronas en la capa oculta, distintos valores de la tasa de aprendizaje y distintos valores del parámetro de exploración-explotación.
En la figura 4.2 se muestran las curvas de aprendizaje (total de recompensas y coste) para distintos valores de profundidad de neuronas de la capa oculta. Como se puede apreciar en dicha figura, no es estrictamente necesario que la capa oculta presente un elevado número de neuronas. Además, si el número de neuronas crece demasiado, como se muestra con la línea roja, se necesitarán aplicar técnicas de regularización. Por lo tanto, para esta tarea se prefiere un valor más pequeño. En el caso de las figuras 4.1 y 4.5 es de cien unidades en la capa oculta.

FIGURA 4.2: Número de neuronas de la capa oculta

La figura 4.3 muestra la evolución de la fase de entrenamiento para variaciones de la tasa de aprendizaje. Se puede apreciar que los valores cuya tasa de aprendizaje es muy pequeña, no llegan a converger de forma estable. Por otro lado tasas muy altas son inestables por lo que en este momento, la tasa correspondiente a $0{,}2$ es un buen valor.

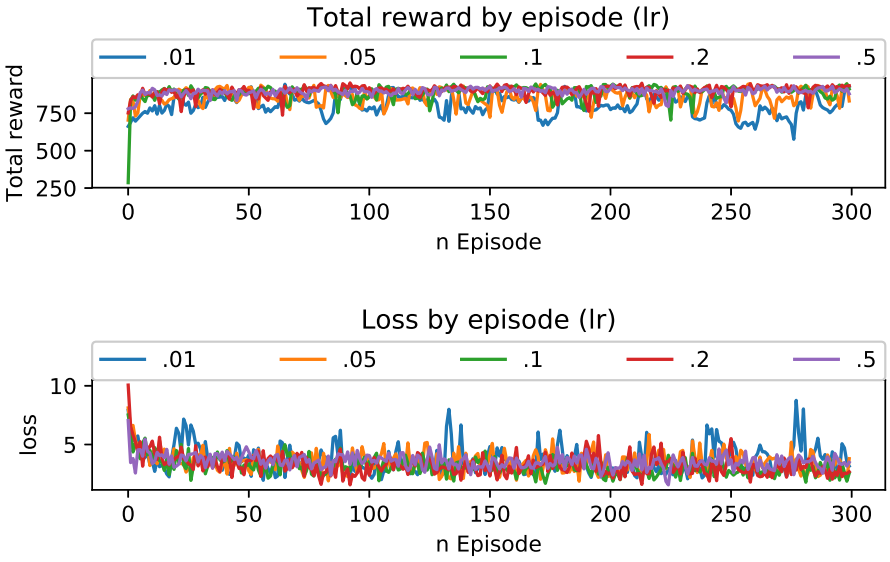

FIGURA 4.3: Variación del entrenamiento dependiendo de la tasa de aprendizaje

En la figura 4.4 se muestran las curvas de aprendizaje para distintas probabilidades

de exploración-explotación. Una tasa de exploración alta únicamente conseguirá resultados aleatorios mientras que una tasa muy pequeña no aprenderá los valores óptimos. En esta figura, se muestra cómo los valores más altos de exploración sesgan los resultados de modo que a mayor exploración (desde un cierto umbral) menor recompensa total. Como valor límite de la exploración con $\epsilon = 1$ se eligen las acciones de forma completamente aleatoria, por lo que la recompensa media total esperada es del $50\,\%$ o en el caso de la figura 4.4, 500 recompensas para un total de 1000 muestras.

FIGURA 4.4: Variación del entrenamiento dependiendo de la exploración-explotación

Una vez representadas distintas gráficas para distintos hiperparámetros, a continuación se obtienen los resultados de pasar el test para la red neuronal entrenada. En este punto se evalúa cómo de bueno es el detector y se alcanzan valores de entorno al $95\,\%$. Este valor parece muy bueno pero como ya se ha comentado, el conjunto de entrenamiento así como el de test utilizado no son del todo correctos. En la figura 4.5 se muestran los resultados de dicho test. En este estudio queda clara la tendencia de los datos. La mayor parte del conjunto pertenece a datos correspondientes con comportamientos de ataques ("1") mientras que una pequeña parte se corresponde con comportamientos normales ("0"). Esto ocasiona que el detector elija con mayor probabilidad los ataques obteniendo un acierto mayor pero no generalizando a posibles nuevas muestras.

FIGURA 4.5: Conjunto de test del detector simple

En la figura 4.6 se muestran los mismo resultados pero con el eje Y en logaritmo. Esto permite apreciar de mejor manera que los falsos positivos son muy bajos comparados con los falsos negativos. Esto implica que el detector se confunde menos asegurando los positivos.

Al algoritmo de *RL* en el caso simple y con el *dataset* propuesto, le es más beneficioso asegurarse de que existe una anomalía que fallar en la detección de un normal. Este problema se trata de solucionar con un *dataset* más equilibrado.

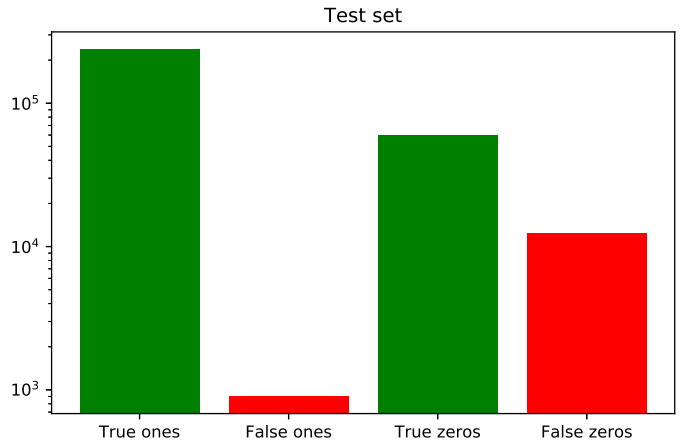

FIGURA 4.6: Conjunto de test del detector simple eje Y logaritmo

4.2. Detector múltiple

El segundo tipo de detector implementado es el que corresponde con detectar explícitamente el tipo de ataque que se envía. Este proceso implica que en vez de detectar únicamente la anomalía, la tenga que categorizar entre todos los posibles ataques disponibles. La estructura de la red neuronal de este detector se corresponde con la figura

3.4. El hecho de separar el conjunto de datos en los dependientes a cada ataque proporciona un conjunto de datos más eficiente. Por otro lado, el número de clases a las que pueden pertenecer estos ataques es demasiado alta y el algoritmo no converge de forma eficiente.

La fase de entrenamiento correspondiente con este tipo de detector se muestra en la figura 4.7, donde se aprecia que el total de recompensas alcanzado difiere mucho del caso anterior. En este detector los mejores valores que se consiguen son de entorno a 750 o un 75 % de aciertos en el conjunto de entrenamiento.

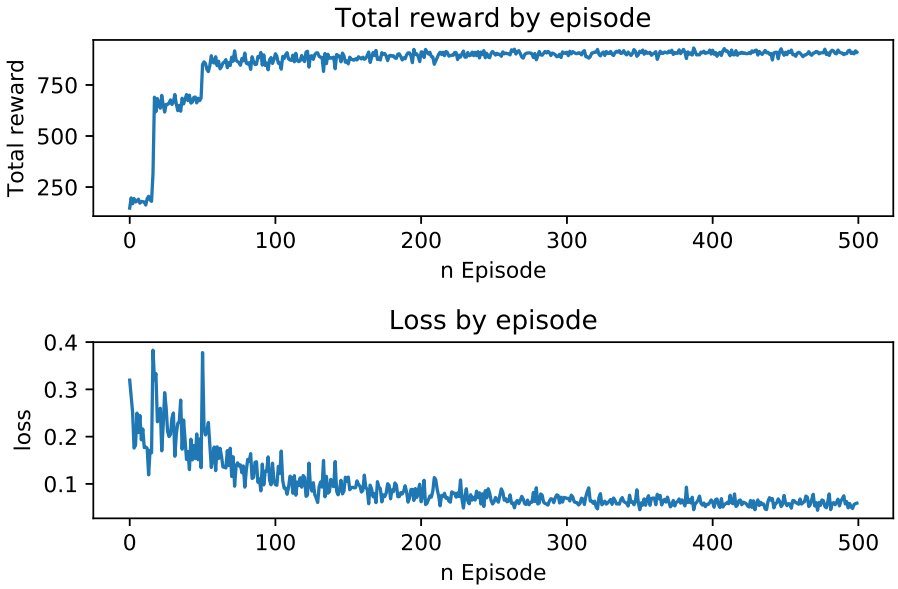

FIGURA 4.7: Fase de entrenamiento para el detector de múltiples ataques

Resulta muy difícil entrenar un clasificador con un número de salidas o ataques tan elevado. El aprendizaje reforzado está más orientado a muestras con alta correlación temporal o que al menos se puedan describir como procesos de *Markov*. En el caso de los detectores el estado define completamente la salida y por lo tanto cumple la condición de falta de memoria, pero no existe ninguna correlación aprovechable entre las muestras. Los resultados de este análisis se muestran en la figura 4.8 donde se puede ver que el resultado parece el esperado. En verde se muestran los ataques que ha estimado de forma correcta, mientras que en rojo se muestran los ataques que ha estimado mal.

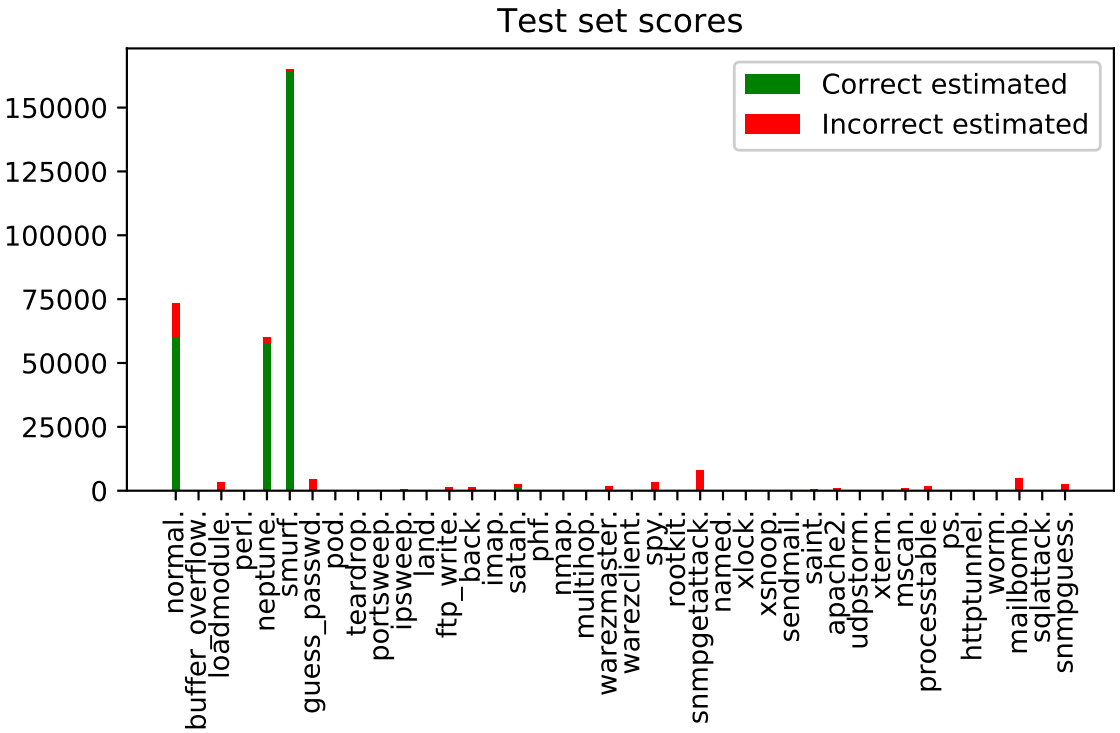

FIGURA 4.8: Conjunto de test del detector múltiple

La figura 4.8 muestra unos resultados con muy poco error. En realidad esto no es así ya que la mayor parte de los ataques se envían con muy poca frecuencia y por lo tanto casi ni se ven. Si por el contrario se utiliza un eje Y logarítmico como el de la figura 4.9 se puede apreciar que en realidad la detección no es tan buena.

La conclusión de estos resultados es que el algoritmo se ha centrado en la detección de los ataques más enviados en la fase de entrenamiento. Esto hace que se detecte el ataque *smurf* y el *neptune* de forma casi perfecta mientras que hay otros ataques que no detecta ni uno solo. El análisis de los fallos que comete este detector se muestran de forma detallada en la tabla 4.1.

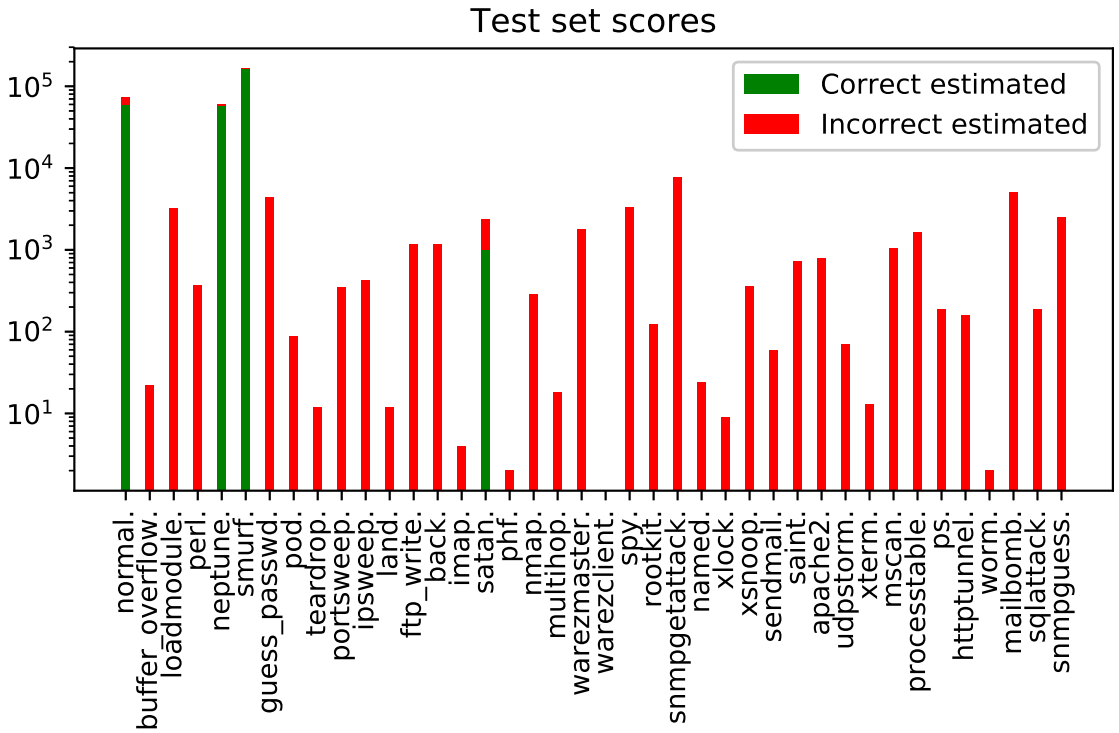

FIGURA 4.9: Conjunto de test del detector múltiple

Ataque	Estimado	Correctos	Fallos	Total
normal	73363	60335	13282	60589
buffer_overflow	0	0	22	22
loadmodule	3251	0	3253	2
perl	368	0	370	2
neptune	60244	57998	2249	58001
smurf	164916	164086	830	164086
guess_passwd	0	0	4367	4367
pod	0	0	87	87
teardrop	0	0	12	12
portsweep	0	0	354	354
ipsweep	117	0	423	306
land	3	0	12	9
ftp_write	1187	0	1190	3
back	77	0	1175	1098
imap	3	0	4	1
satan	1723	989	1378	1633
phf	0	0	2	2
nmap	204	0	288	84
multihop	0	0	18	18
warezmaster	188	0	1790	1602
warezclient	0	0	0	0
spy	3355	0	3355	0

Ataque	Estimado	Correctos	Fallos	Total
rootkit	109	0	122	13
snmpgetattack	101	0	7842	7741
named	7	0	24	17
xlock	0	0	9	9
xsnoop	361	0	365	4
sendmail	43	0	60	17
saint	0	0	736	736
apache2	1	0	795	794
udpstorm	68	0	70	2
xterm	0	0	13	13
mscan	0	0	1053	1053
processtable	882	0	1641	759
ps	174	0	190	16
httptunnel	0	0	158	158
worm	0	0	2	2
mailbomb	6	0	5006	5000
sqlattack	189	0	191	2
snmpguess	80	0	2486	2406

TABLA 4.1: Análisis detallado del test detector múltiple

4.3. Detector del tipo

El último detector implementado se corresponde con el detector de tipo de ataque. Este detector tiene como ventaja un menor número de clasificaciones. Un total de cinco clases, cuatro para ataques y otro más para el tipo normal. La estructura de la red neuronal para este caso se corresponde con la de la figura 3.5.

En la figura 4.10 se muestra la gráfica de entrenamiento de este detector. Se observa que la recompensa total por episodio en este caso aumenta respecto del anterior, obteniendo una recompensa media final de entorno al $98\,\%$. Este valor se corresponde con un valor muy bueno teniendo en cuenta que el detector está menospreciando los valores de los ataques que menos suceden. Esto es un gran problema como se comentó en el apartado 3.2.1 ya que justamente los ataques que menos aparecen son los más problemáticos. Se hace necesario un conjunto de datos en el que la cantidad de los ataques esté más equitativamente equilibrada.

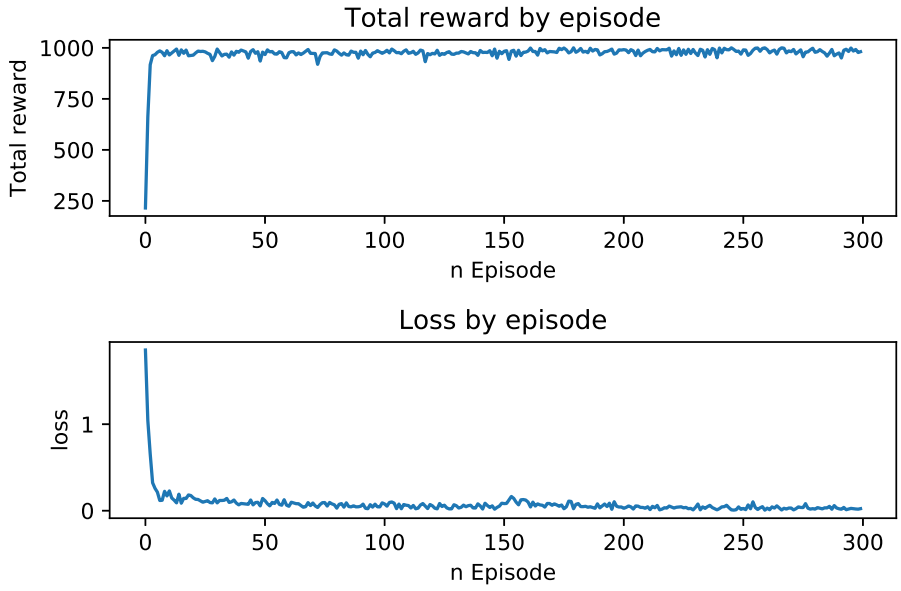

FIGURA 4.10: Fase de entrenamiento para el detector del tipo de ataque

En la figura 4.11 se muestra de forma gráfica el test realizado utilizando el detector de tipo. En verde se presentan las muestras correctamente clasificadas mientras que en rojo las incorrectas. Se puede apreciar que mejora notablemente el realizado con el detector múltiple. En la tabla 4.2 se muestra el análisis detallado de este detector, donde se puede ver que la mayoría de las detecciones están centradas en los tipos más probables.

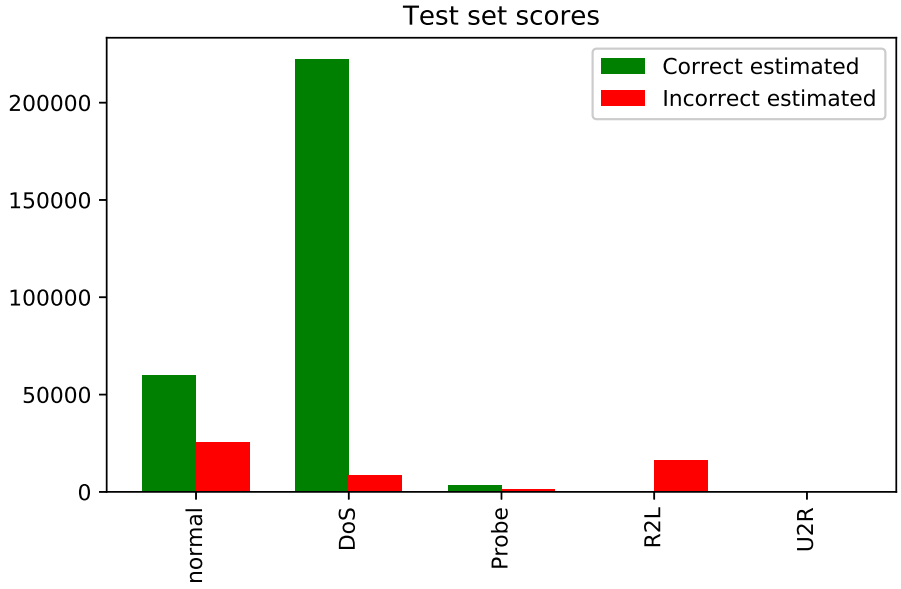

FIGURA 4.11: Fase de entrenamiento para el detector del tipo de ataque

Tipo	Estimadas	Correctas	Fallos	Total
normal	84411	59873	25257	60592
DoS	222971	222235	8346	229845
Probe	3638	3261	1282	4166
R2L	0	0	16189	16189
U2R	0	0	228	228

TABLA 4.2: Análisis detallado del test detector de tipo

Capítulo 5

Mejoras

En la implementación de los algoritmos de aprendizaje reforzado siempre existen ciertos problemas o barreras que estropean el funcionamiento de los algoritmos. Muchos de estos problemas son comunes en la mayoría de mecanismos de *machine learning* como la correlación entre las muestras. Por otro lado, también se encuentran problemas que solo afectan al aprendizaje reforzado. Entre estos últimos se tienen problemas de estabilidad, problemas de exploración y explotación del conocimiento así como dificultad de convergencia. Estos temas se tratan con detalle en el articulo publicado por **DeepMind** sobre control a nivel humano aplicando aprendizaje reforzado profundo [7]. Como primera propuesta de mejora de la detección se plantea la modificación del problemático *dataset*. Con esta modificación se pretende dar solución a la redundancia del original. La segunda modificación se corresponde con incluir ciertas tecnologías aplicadas en el documento de **DeepMind** [7]. Entre otras, se implementan "*experience replay*", "*target network*" y "*Huber loss*". Por último se plantea como propuesta la implementación de un algoritmo contrincante. Este contrincante pretende hacer uso de dos "maquinas" enfrentadas con el fin de conseguir el mejor entrenamiento posible para ambas.

5.1. Cambio de dataset

La primera modificación necesaria para tratar los problemas de detección citados en el capítulo 4 es el cambio de *dataset*. Esta modificación va a permitir un entrenamiento de la red neuronal con menor sesgo y por lo tanto una función de aproximación más precisa a cualquiera de los ataques (generalización).

El *dataset* NSL-KDD soluciona gran parte de los problemas descritos. Se puede encontrar un análisis detallado en [18] donde se realizan estudios estadísticos de la cantidad de datos redundantes. La tabla 5.1 muestra los resultados de las estadísticas analizadas. En esta tabla se puede apreciar como la mayor parte de los datos del conjunto original son redundantes y se pueden reducir.

En [19] se puede encontrar un análisis de diferentes algoritmos de *machine learning*, así como ciertas técnicas de reducción de *features*, que pueden ser muy útiles a la hora de mejorar la detección.

	Datos originales	Datos distintos	Ratio de reducción
Ataques	3925650	262178	93.32 %
Normal	972781	812814	16.44 %
Total	4898431	1074992	78.05 %

TABLA 5.1: Estadísticas de los datos redundantes en el conjunto de datos de entrenamiento del *KDD*

5.2. Experience Replay

La memoria de "experiencia" se introduce en [7] con intención de solucionar la divergencia de las redes neuronales profundas causado por el aprendizaje *online*. Esta memoria de experiencia o "*experience reaplay*" almacena la experiencia del agente en cada instante temporal, $e_t = (s_t, a_t, r_t, s_{t+1})$, en un nuevo conjunto de datos $D_t = \{e_1, \ldots, e_t\}$, agrupados a lo largo de muchos episodios (donde el final de un episodio se produce cuando se alcanza un estado terminal) en una repetición de memoria. Esta mejora tiene significantes ventajas frente al aprendizaje online del algoritmo *Q-learning*. Primero, cada paso de experiencia es utilizado en la actualización de múltiples pesos, lo que incrementa la eficiencia de los datos. En segundo lugar, el aprendizaje de muestras consecutivas es ineficiente, si se tiene en cuenta la alta correlación entre las muestras, por lo que si se aleatorizan las muestras se rompe esta correlación lo que reduce la varianza en las actualizaciones. Por último, esta memoria permite evitar caer en bucles de comportamiento que producen un mínimo local pobre muy probables con el aprendizaje basado en políticas.

La implementación de la memoria puede suponer un coste de recursos importante. En los casos en los que la memoria tenga que almacenar imágenes hay que tener mucho cuidado con el tamaño total que esto supone ya que el uso de memoria RAM es muy elevado. En el caso del detector implementado es más difícil que se llene la memoria RAM ya que el dataset de entrenamiento *KDD* completo y formateado ocupa hasta 2 GB en el caso del detector múltiple. Teniendo en cuenta esto, es importante dimensionar correctamente la memoria para no exceder demasiado el consumo de RAM. La memoria, por lo tanto, almacena N muestras de datos como un conjunto de memoria finita. El problema de este tipo de memorias es que no almacenan implícitamente las transiciones más importantes, sino que a medida que van llegando se van almacenando las nuevas y eliminando las viejas cuando se llena la memoria. Un sistema más sofisticado que mejora el actual podría enfatizar las transiciones que producen más información, similar a un barrido priorizado. Dentro de este tipo se sitúa lo que se conoce como "Prioritized Experience Replay".

A continuación se muestra una posible implementación de la memoria:

```
1 class ReplayMemory(object):
2     """Implements basic replay memory"""
3
4     def __init__(self, observation_size, max_size):
5         self.observation_size = observation_size
6         self.num_observed = 0
7         self.max_size = max_size
8         self.samples = {
```

```
 9             'obs'      : np.zeros(self.max_size * 1 \
10             * self.observation_size,
11             dtype=np.float32).reshape(self.max_size,
12             self.observation_size),
13
14             'action'   : np.zeros(self.max_size * 1, dtype=np.int16).
                  reshape(self.max_size, 1),
15
16             'reward'   : np.zeros(self.max_size * \
17             1).reshape(self.max_size, 1),
18
19             'terminal' : np.zeros(self.max_size * 1, dtype=np.int16).
                  reshape(self.max_size, 1),
20                }
21
22     def observe(self, state, action, reward, done):
23         index = self.num_observed % self.max_size
24         self.samples['obs'][index, :] = state
25         self.samples['action'][index, :] = action
26         self.samples['reward'][index, :] = reward
27         self.samples['terminal'][index, :] = done
28
29         self.num_observed += 1
30
31     def sample_minibatch(self, minibatch_size):
32         max_index = min(self.num_observed, self.max_size) - 1
33         sampled_indices = np.random.randint(max_index, size=minibatch_size)
34
35         s       = np.asarray(self.samples['obs'][sampled_indices, :], dtype=
                    np.float32)
36         s_next = np.asarray(self.samples['obs'][sampled_indices+1, :],
                    dtype=np.float32)
37
38         a       = self.samples['action'][sampled_indices].reshape(
                    minibatch_size)
39         r       = self.samples['reward'][sampled_indices].reshape((
                    minibatch_size, 1))
40         done    = self.samples['terminal'][sampled_indices].reshape((
                    minibatch_size, 1))
41
42         return (s, a, r, s_next, done)
```

5.3. Target network

Uno de los problemas principales de los algoritmos de *RL* es la dificultad de convergencia. La idea de utilizar *Q-learning* implica tener que optimizar la ecuación de actualización de la función de valor Q (2.29) de forma recursiva. Esto implica que con cada actualización, el valor de $Q(s, a)$ y de $Q(s', a')$ se actualizan a la vez. De este modo en cada iteración la red se aproxima al *target* mientras que este mismo se desplaza, haciendo más difícil así la convergencia. Como solución a este problema se puede hacer uso de dos redes neuronales, una para realizar la optimización de la función y otra para el *target*. Así se inicializan dos redes idénticas y se optimiza la función del modelo dejando fijos los parámetros de la red correspondiente con el *target*. Cada un cierto número de iteraciones se copia la red optimizada como nuevo *target* y se vuelve

a optimizar realizando este proceso de forma cíclica. Se puede encontrar información detallada de este procedimiento en [20].

Las modificaciones respecto del algoritmo de actualización normal 3.2 se muestran a continuación:

```
next_actions = []
# Predicción de la función de valor siguiente
Q_prime = self.model_network.predict(next_states, self.minibatch_size)
# Mejores siguientes acciones para actualización en batch
for row in range(Q_prime.shape[0]):
    best_next_actions = np.argwhere(Q_prime[row] == np.amax(Q_prime[row]))
    next_actions.append(best_next_actions[np.random.choice(len(
        best_next_actions))].item())
sx = np.arange(len(next_actions))
# Obtengo la TARGET para actualizar la red
Q = self.target_model_network.predict(states, self.minibatch_size)
# Q-learning update
# target = reward + gamma * max_a'{Q(next_state, next_action))}
targets = rewards.reshape(Q[sx, actions].shape) + \
          self.gamma * Q[sx, next_actions] * \
          (1-done.reshape(Q[sx, actions].shape))
Q[sx, actions] = targets

# Calculo la función de perdidas y actualizo el modelo principal
loss = self.model_network.model.train_on_batch(states,Q)#inputs, targets

# Cada X tiempo actualizo el modelo de la red TARGET
# timer to ddqn update
self.ddqn_update -= 1
if self.ddqn_update == 0:
    self.ddqn_update = self.ddqn_time
    self.target_model_network.model.set_weights(self.model_network.model.
        get_weights())
```

5.4. Huber Loss

En el artículo de *DQN* [7] se describe la forma de recortar la función de error entre -1 y 1. Debido a que la función de pérdidas de valor absoluto tiene una derivada de -1 para todos los valores negativos de x y una derivada de $+1$ para todos los valores positivos de x, recortar la función de error cuadrática entre -1 y 1 corresponde a utilizar una función de pérdida de valor absoluto para errores fuera del intervalo $(-1, 1)$. Esta forma de recortar el error proporciona al algoritmo mayor estabilidad.

No se debe interpretar de mal manera la forma de recortar la función. No es un simple recorte de la función cuadrática en el intervalo $(-1, 1)$, esto ocasiona pérdidas de cero cuando los errores son muy altos y es precisamente el efecto contrario. La función de pérdida que se debe usar es también conocida como *Huber loss*. Esta función representa una aproximación cuadrática cuando los errores son pequeños por lo tanto un tramo de progresión suave mientras que presenta un comportamiento lineal cuando se supera un cierto umbral de error. Esto permite que los algoritmos de *RL* que tienen grandes problemas de convergencia lleguen a un estado estacionario de forma más rápida. La ecuación 5.1 describe esta progresión, siendo el parámetro δ el valor en el que

se recorta dicha función. Según el artículo [7], este recorte debe producirse con el valor de $\delta = 1$, pero puede ser un parámetro a modificar más adelante.

$$\text{Huber}(a) = \begin{cases} \frac{1}{2}a^2, & \text{si } |a| \leq \delta. \\ (|a| - \frac{1}{2}\delta), & \text{resto.} \end{cases} \tag{5.1}$$

En la figura 5.1 se muestran la función de costes de *Huber* junto con la cuadrática y la de valor absoluto. En el ejemplo el parámetro que modifica el comportamiento de cuadrático a lineal (*clip*) tiene el valor de $\delta = 2$ para poder visualizar un poco mejor el efecto de recorte. Como ya se ha comentado, en la implementación del detector este valor será de $\delta = 1$.

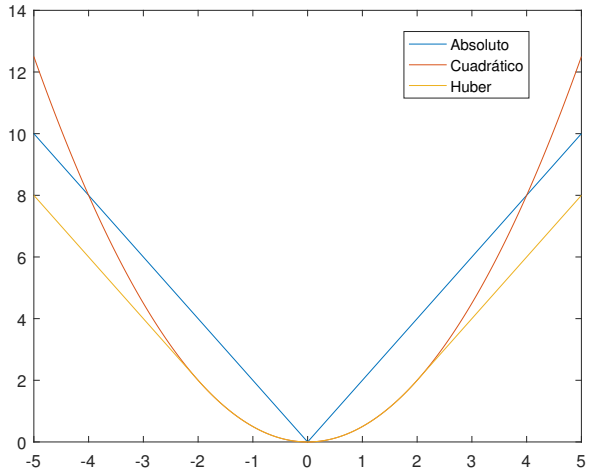

FIGURA 5.1: Distintas funciones de coste

5.5. Resultados con las mejoras *DDQN*

Las mejoras explicadas en este capítulo únicamente van a ser aplicadas al detector de los distintos tipos de ataque. Este último detector es más completo que el detector simple y más preciso que detectar cada ataque en especifico. Como posible implementación de una defensa ante los distintos ataques, es suficiente con conocer el tipo, por lo que es suficiente con aplicar los cambios a este detector.

En la figura 5.2 se muestran las curvas de entrenamiento del detector con las mejoras realizadas. En este caso, la curva parece similar a la de la figura 4.10, pero ahora ya no se están sobre-estimando los valores repetidos del *dataset*. Esto implica que los resultados sobre el conjunto de test van a ser más equilibrado. Es posible que todas las mejoras que se aplican a *DDQN* no son del todo necesarias ya que en este caso la correlación entre muestras es nula pero se ha intentado mantener la notación del término *DDQN* y se han implementado como se detalla en [20, 21].

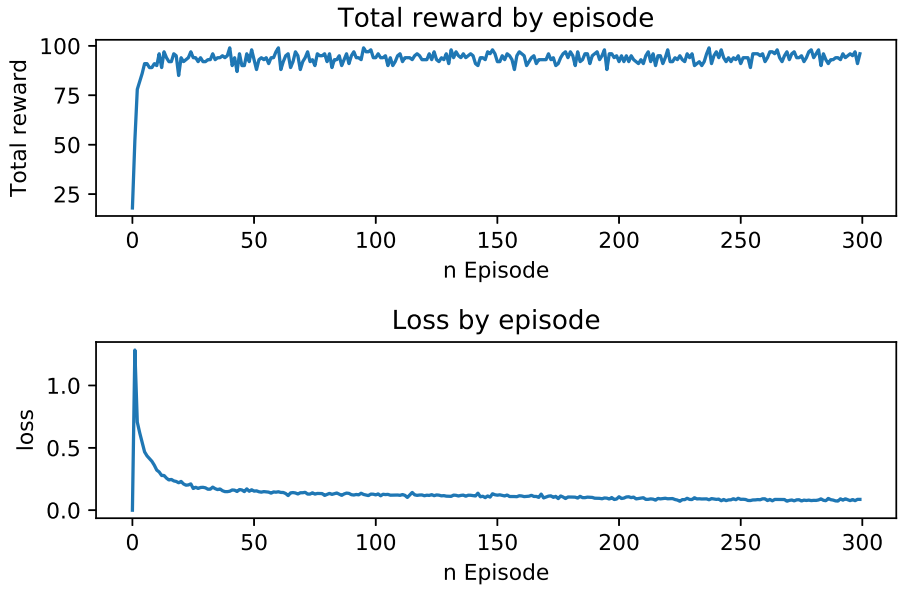

FIGURA 5.2: Entrenamiento del detector con las mejoras *DQN*

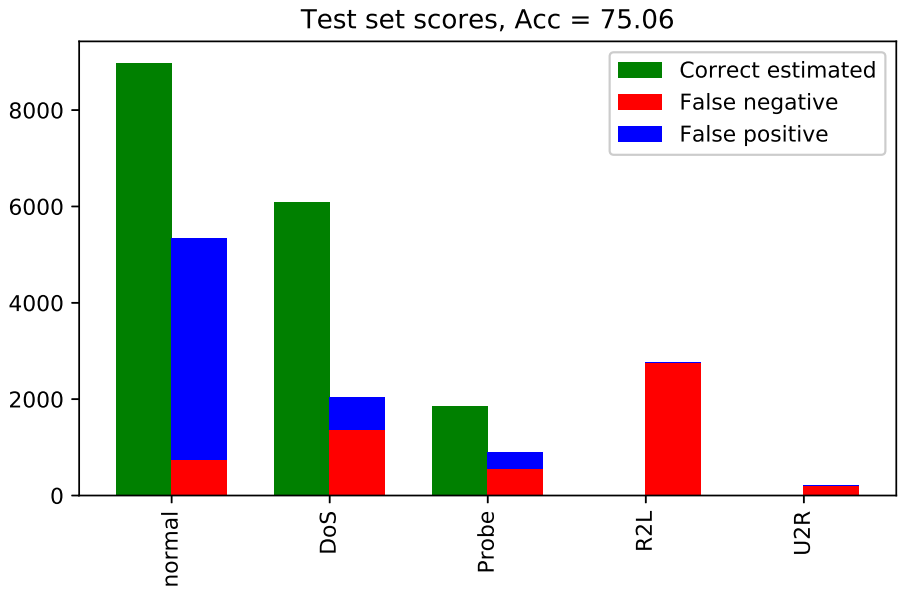

FIGURA 5.3: Test del detector de tipo mejorado

Los resultados, correspondientes con las mejoras *DDQN*, se muestran en la figura 5.3. En este caso, la tasa de detección es menor que cuando no se había mejorado el dataset, pero la detección es más fiable. En total, se consigue una detección de entorno al 75 %, teniendo en cuenta que tanto *R2L* como *U2R* no obtienen ninguna detección. El detector se sigue centrando en detectar los ataques más repetidos como se puede ver en las estimaciones correctas (color verde). Por otro lado, el número de falsos positivos que se dan en la clase normal es sumamente elevado (color azul). Prácticamente todos los falsos negativos (color rojo) que fallan en la clasificación de los ataques *R2L* y *U2R*

se clasifican de manera errónea dentro de esta clase. Esta figura representa por lo tanto los fallos agrupados en dos colores, rojo y azul. Estos fallos significan en parte lo mismo ya que la suma de los rojos tiene que ser igual a los azules, pero dan una perspectiva de en que clase se están concentrando los fallos. Gracias a esta representación se muestra de forma clara que la clase normal está teniendo la mayor parte de los flujos correctos detectados pero por el contrario esta malinterpretando ataques como clases normales que pueden suponer graves situaciones de seguridad.

En la tabla 5.2 se muestra en detalle los resultados del conjunto de test. En esta tabla se puede apreciar de forma detallada los aciertos que comete el detector. Es significante detallar que en el caso de la clase normal los aciertos del total cubren un $93\,\%$ de exactitud, el problema reside en que se estiman 13532 sobre un total de 9712 y únicamente se aciertan 9004. Como ya se ha explicado, muchas de las muestras que deberían estar clasificadas en *R2L* o *U2R* se clasifican incorrectamente en la clase normal. La clasificación errónea supone una equivocación en cuanto a porcentajes pero es más importante sobre todo si ataques tan peligrosos como *R2L* y *U2R* se clasifican de forma incorrecta dentro de la clase normal. Si ocurriese lo contrario (que un flujo normal se clasifique como ataque de cualquier tipo) no sería tan preocupante, pero en este caso sí. Se podría resolver este dilema aplicando una asignación de recompensas diferente dependiendo de la equivocación. En cuanto a las clases *DoS* y *Probe* se encuentran en un caso parecido de sobre clasificación, debida al mismo problema de clasificación de los otros dos tipos.

Tipo	Estimadas	Correctas	Fallos	Total
normal	13612	9035	5253	9712
DoS	6722	6215	1750	7458
Probe	2210	1765	1101	2421
R2L	0	0	2753	2753
U2R	0	0	200	200

TABLA 5.2: Resultados del detector

5.6. Arquitectura *Dueling Network*

La arquitectura de *Dueling Network* [22], se corresponde con una mejora más al complejo de mejoras del algoritmo *DDQN*. En este caso, la función de valor Q se puede descomponer en dos funciones más sencillas. Para ello se descompone en la función de valor de un estado $V(s)$. Esta función representa el valor único que se puede llegar a alcanzar en un estado sin tener en cuenta las acciones que posteriormente se tomarán. La representación de esta función se muestra a la izquierda de la figura 2.5 para una política π y a la izquierda figura 2.6 para su valor óptimo. Este valor indica el cómo de bueno es estar en un estado. La otra función en la que se descompone la función Q es en la función de ventaja o *advantaje function* $A(a)$ que ofrece información relativa a la cantidad de recompensa que puede producir tomar una acción u otra. En conjunto, estas dos funciones componen la función **Q**, que se puede despejar de la ecuación (2.44). Para ello se necesita obtener de forma previa el valor de la estimación de la

función de *advantage* y en ello consiste la separación de las redes. La función **Q** por lo tanto se obtiene de la siguiente forma:

$$Q(s,a) = V(s) + A(a) \tag{5.2}$$

El objetivo de *Dueling DQN* es tener una red que calcule por separado las funciones de *advantage* y valor, y las combine de nuevo en una sola función Q en la capa final. Intuitivamente, la arquitectura de *dueling* puede aprender qué estados son (o no) valiosos, sin tener que aprender el efecto de cada acción para cada estado. Esto es particularmente útil en los estados en los que sus acciones no afectan al entorno de ninguna manera relevante. En los experimentos realizados en [22] se demuestra que esta arquitectura es capaz de aprender de forma más rápida las acciones óptimas incluso cuando se añaden al sistema acciones redundantes de prueba. La representación de esta estructura se muestra de forma clara en la parte inferior de la figura 5.4 mientras que la superior se corresponde con la estructura normal.

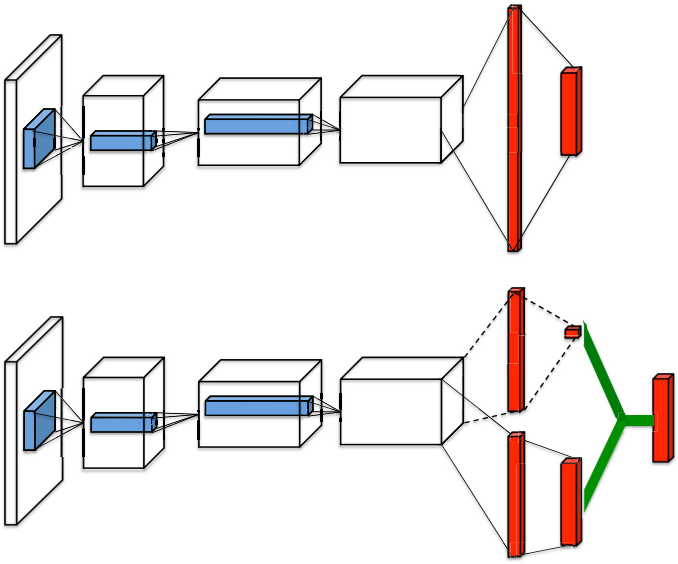

FIGURA 5.4: Arquitectura normal superior y *Dueling Network* inferior

La implementación de esta estructura consta de una capa de entrada con el número de elementos igual al espacio de observación de los estados, dos capas ocultas de cien unidades para ofrecer una alta profundidad de no linealidades y una tercera capa oculta. Esta última capa oculta se divide en dos partes representativas de la función de *advantage* y la función de valor. Una vez se dispone de esas dos funciones, se calcula el valor de la función **Q** como la ecuación 5.2. Para mejorar la estabilidad del algoritmo y reducir la varianza en la medida de lo posible, se procede a restar el valor medio de la función de *advantage*. La red que ofrece estos resultados se compone de la siguiente manera:

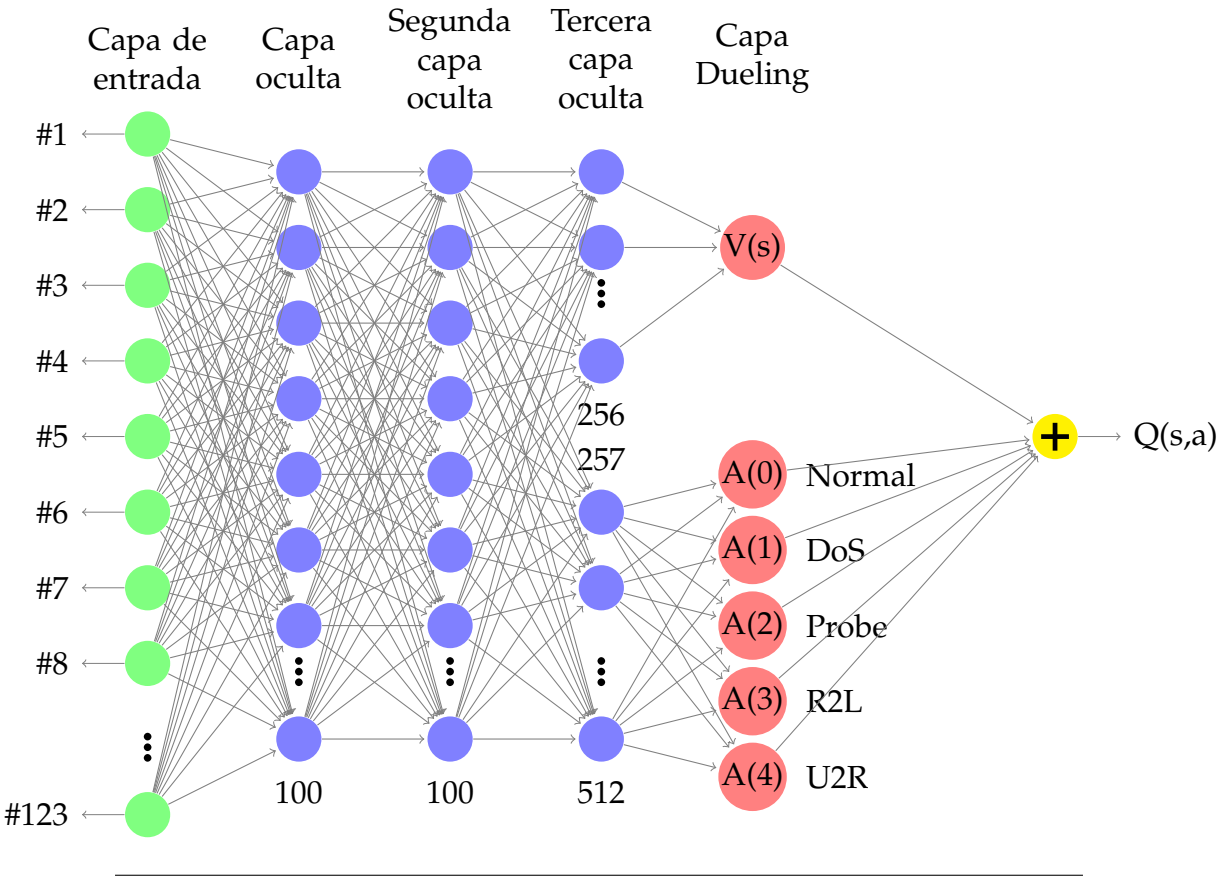

FIGURA 5.5: Estructura de la red neuronal *dueling*

Dado que la implementación de esta red se realiza mediante tensorflow, el fragmento de código necesario para la realización es muy distinto al correspondiente código de actualización de **Q** (3.2). A continuación se muestra un fragmento representativo del código utilizado para la representación de la función de valor:

```
# Definición del placeholder para los datos de entrada
Input =  tf.placeholder(shape=[None,env.obs_size],dtype=tf.float32)
# Creación de las tres capas ocultas
out1 = layer.fully_connected(inputs=self.Input,num_outputs=100)
out2 = layer.fully_connected(inputs=self.out1,num_outputs=100)
out3 = layer.fully_connected(inputs=self.out2,num_outputs=split_size)
# split_size = profundidad de la capa que se va a dividir
# Se divide la red en dos partes iguales
streamAC,streamVC = tf.split(out3,2,1)
# Se utiliza xavier init para optimizar la inicialización de pesos
xavier_init = tf.contrib.layers.xavier_initializer()
# Se preparan las variables a optimizar con su tamaño correspondiente
AW = tf.Variable(xavier_init([split_size//2,env.num_actions]))
VW = tf.Variable(xavier_init([split_size//2,1]))
# Se multiplica la salida de las redes por las variables o pesos
Advantage = tf.matmul(streamAC,AW)
Value = tf.matmul(streamVC,VW)
# Se calcula la función Q
Qout = Value + tf.subtract(Advantage,tf.reduce_mean(Advantage,axis=1,
    keepdims=True))
```

En la figura 5.6 se muestra el proceso de entrenamiento de esta red. Se puede apreciar que los resultados son similares al resto de entrenamientos. En la figura 5.7 se muestran los resultados de esta red, donde se puede apreciar el comportamiento esperado. Las estimaciones correctas se muestran en color verde y se observa que categoría de "normal" se estima de forma casi perfecta acertando un $97{,}24\,\%$ de las muestras. Por otro lado, en esta misma clase se asignan la mayor parte de los fallos de *R2L* o incluso alguno de *DoS*. Estos fallos en la categorización se corresponden con el segmento azul siendo estimaciones en la clase normal que deberían estar en otra clase. En rojo se muestran las detecciones que no se han realizado de la misma clase, por este motivo el fragmento rojo de la clase normal se corresponde con el $2{,}76\,\%$ restante. Como en los detectores anteriores, la clase *R2L* se muestra casi al completo en rojo debido a que en esa clase no se han clasificado flujos de otras sino que los fallos se deben por completo a faltas de clasificación o falsos negativos.

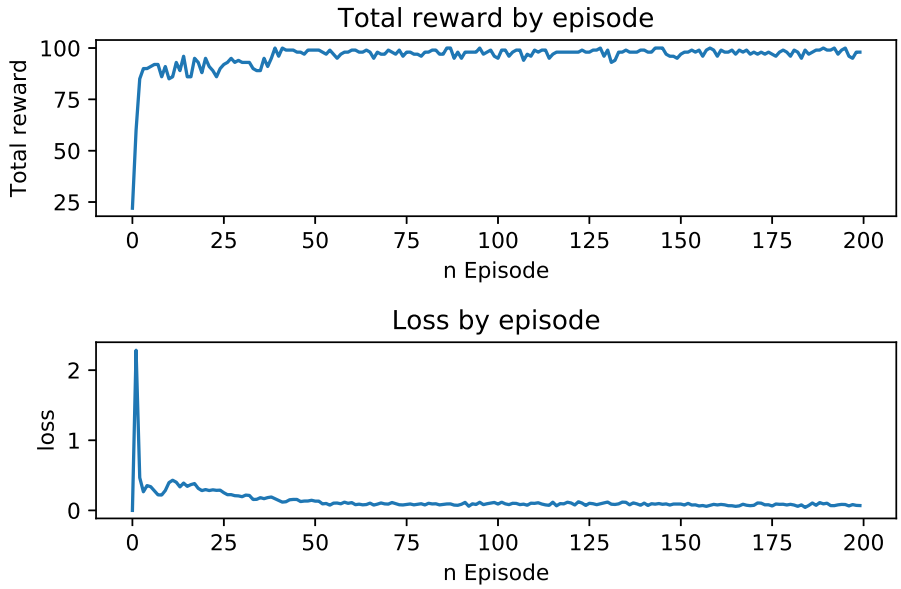

FIGURA 5.6: Entrenamiento de la red *Dueling Network*

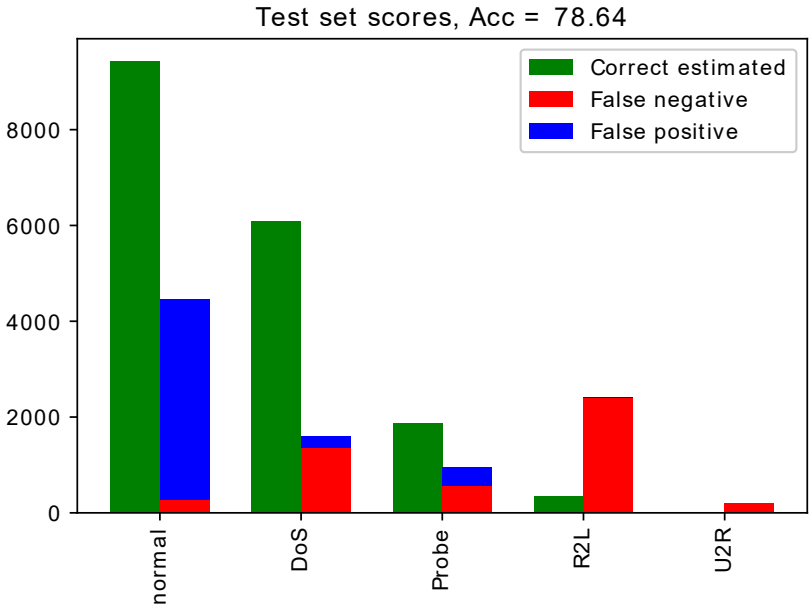

FIGURA 5.7: Test de la red *Dueling Network*

En la tabla 5.3 se muestran los resultados del test detallados para la estructura de red *dueling*. Como es de esperar, la mayor parte de las detecciones se centra en las clases de normal, *DoS* y *Probe*. Sin embargo este detector es capaz de detectar un mejor porcentaje de muestras *R2L* que los métodos anteriores. Esta mejora puede deberse a la propia separación del valor de las acciones al valor del estado ya que en este caso el valor de las acciones representa casi por completo el valor real de la función.

Tipo	Estimadas	Correctas	Total	Aciertos %
normal	13710	9443	9711	97.24
DoS	6448	6212	7458	83.29
Probe	2213	1848	2421	76.33
R2L	361	349	2754	12.67
U2R	0	0	200	0

TABLA 5.3: Resultados del detector

5.7. Multi-agent Competition Reinforcement Learning

Para poder maximizar el proceso de detección de los ataques se pretende implementar un nuevo punto de vista. En esta situación existen dos sistemas contrincantes en el que cada uno intentará maximizar su recompensa por medio del algoritmo *RL*. De este modo el atacante pretende elegir el tipo de ataque que está siendo más efectivo mientras que el defensor mejorará sus propiedades de generalización.

En la figura 5.8 se muestra un esquema del proceso de este entorno. En este caso existen dos agentes, uno encargado de preparar los ataques y otro encargado de la defensa. El proceso es el siguiente:

1. El entorno muestra al atacante un estado, en primer lugar de forma aleatoria.
2. El atacante elige un nuevo ataque basándose en su política ($39 + 1$ ataques posibles). Este ataque tendrá asociado un estado correspondiente a esa etiqueta. De entre todos los ataques con la misma etiqueta, se selecciona uno de forma aleatoria.
3. El entorno entrega este estado a la defensa.
4. La defensa elige las acciones que maximizan su recompensa atendiendo a los estados recibidos. Esta acción se corresponde con el tipo de ataque se el detector estima haber recibido.
5. Teniendo en cuenta las acciones elegidas por el atacante y las elegidas por la defensa, el entorno devuelve las recompensas para que los agentes atacante y defensa aprendan de la interacción.

FIGURA 5.8: Descripción del entorno y los distintos agentes

5.7.1. Agente atacante

El agente atacante de este entorno es el responsable de obtener el mayor número de ataques no detectados posible. En esta condición, el agente que realiza el ataque escoge una acción entre todos los ataques disponibles que se muestran en la tabla 3.5. Para ello, la red neuronal que aproxima la función de valor se corresponde con la figura 3.4. Una vez se obtienen los valores de las acciones a enviar, el entorno es el encargado de buscar los estados correspondientes a estas acciones. El entorno realiza una extracción aleatoria del dataset que se corresponda con la acción enviada.

5.7.2. Agente defensor

En la oposición, el agente defensor es el responsable de obtener la mayor recompensa posible detectando de forma correcta los ataques enviados. Con el fin de implementar el detector únicamente para conocer los tipos de ataques enviados, la red neuronal

que aproxima la función de valor en este caso se corresponde con la figura 3.5. Las acciones elegidas por el defensor, son por lo tanto los distintos tipos de ataques.

5.7.3. Recompensas

En este entorno, las recompensas asociadas a las acciones se corresponden con las siguientes:

- Recompensa positiva $+1$ para el atacante si la acción de la defensa no coincide con la acción del atacante.
- Recompensa positiva $+1$ para la defensa si la acción elegida por la defensa coincide con la del ataque.
- Recompensa de $+0$ para el que se equivoca en la decisión.

Esta asignación de recompensas se corresponde con la formulación más sencilla posible. Existen otras posibilidades de asignación de las recompensas dependiendo del tipo de ataque enviado y la defensa estimada. No todos los tipos de ataques son igual de dañinos por lo que se pueden definir recompensas en función a un parámetro de peligrosidad de los ataques.

5.7.4. Resultados *Adversarial*

En la figura 5.10 se muestra la fase de entrenamiento del entorno competitivo. Para que el entorno sea productivo y beneficioso para la tarea que se pretende conseguir, el agente que se encarga del ataque no puede ser igual de "inteligente" que el agente de la defensa. En el caso de ser iguales, el ataque va a alternar completamente la decisión de sus ataques de forma que intente averiguar cual son los ataques que producen un fallo en la defensa. Este hecho no va a producir mejoras en el algoritmo de detección ya que incluso puede ocasionar que la defensa converja en valores erróneos. Para evitar estos posibles errores, se limita la explotación del conocimiento del agente encargado del ataque de forma que se pueda asegurar que la defensa converge a un estado estacionario correcto. Para ello, el ataque escoge las acciones del siguiente modo:

$$Ataque = \begin{cases} \text{argmax}(Q) \text{ con } \mathcal{P} = p_0 \\ random \text{ con } \mathcal{P} = 1 - p_0 \end{cases} \tag{5.3}$$

De la ecuación de probabilidad del agente atacante (5.3) se puede determinar que al menos con una probabilidad p_0 se escogen los ataques que más perjudican a la defensa, y es con éstos con los que se consigue mejorar la detección. Del mismo modo que para la defensa se decide implementar una política de ϵ-*greedy* con decaimiento, por lo que ambos agentes comienzan con una exploración del $100\,\%$, lo que posibilita la exploración inicial para maximizar el aprendizaje. Mientras que el agente de la defensa experimenta un decaimiento de la exploración hasta valores del $1\,\%$ o incluso menores, el agente del ataque recorta el valor mínimo de exploración. La elección de este mínimo se puede determinar de forma sencilla realizando un barrido paramétrico como el que se muestra en la figura 5.9. Los valores no varían mucho pero se puede ver que cuando la exploración decrece a valores muy pequeños la precisión disminuye considerablemente.

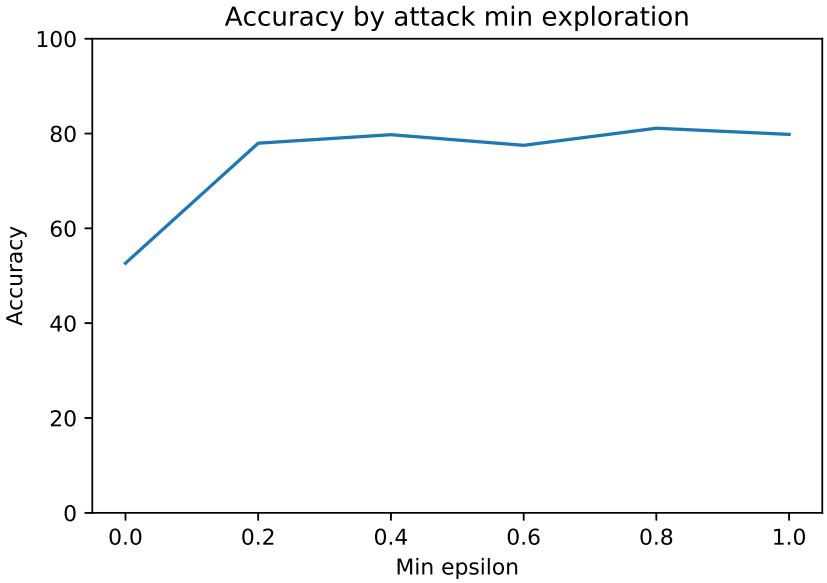

FIGURA 5.9: Ajuste del parámetro de mínima exploración

Una vez se escoge un valor apropiado para el mínimo de exploración del ataque se procede al entrenamiento de este sistema. Los resultados del proceso de entrenamiento de este detector adversario se muestran en la figura 5.10. Como se puede apreciar, en este caso, los resultados son completamente distintos al caso anterior siendo la defensa la mayor beneficiada. Con esta modificación en el corte mínimo de exploración ($1 > \epsilon > p_0$) se consiguen unos resultados de en torno al $80{,}5\,\%$, por lo tanto una mejora significativa respecto al detector *DDQN*. Para la obtención de estos resultados se ha utilizado una red neuronal sin arquitectura *Dueling* por su mayor simplicidad, no obstante si se modifica se podrían alcanzar valores superiores. La intención de estudiar el entorno competitivo es la de estudiar las mejoras respecto al algoritmo *Q-learning* básico. Por otro lado se pueden combinar las mejoras detalladas en un unico proceso.

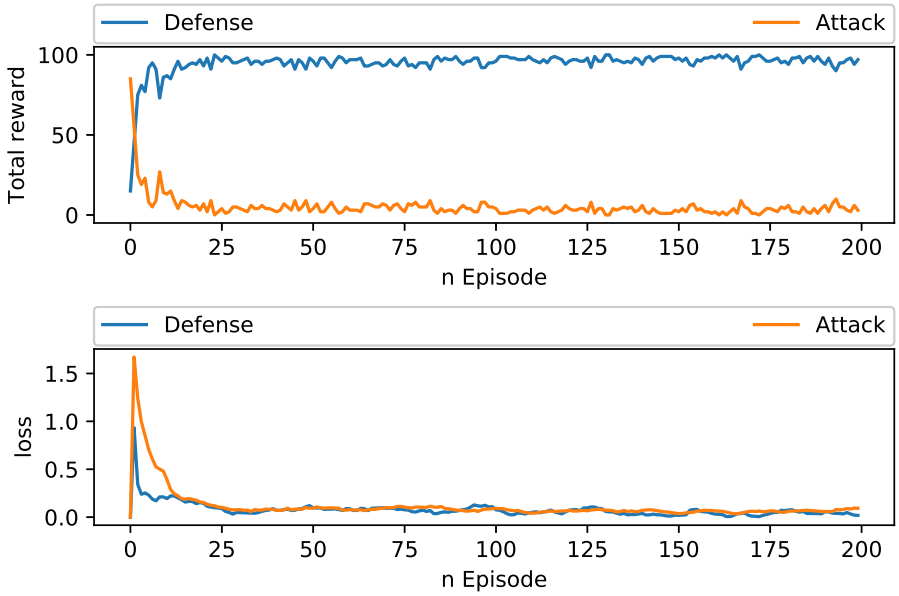

FIGURA 5.10: Entrenamiento competitivo con mínimo de exploración

En la figura 5.11 se muestra de forma gráfica los resultados del test y en la tabla 5.4 se encuentran los ataques detallados. Se ha considerado interesante incluir los resultados con el *F1 score* ya que proporciona información relativa no solo a los correctamente estimados sino a los falsos positivos, falsos negativos, verdaderos positivos y verdaderos negativos. En este caso, la tasa de detección de *R2L* y *U2R* es positiva. Además, los falsos positivos de *R2L* y *DoS* son muy bajos por eso el *F1 Score* de *DoS* es tan alto. Por otro lado, la clase normal sigue sobreestimandose añadiendo a esta clase la mayor parte de fallos del resto.

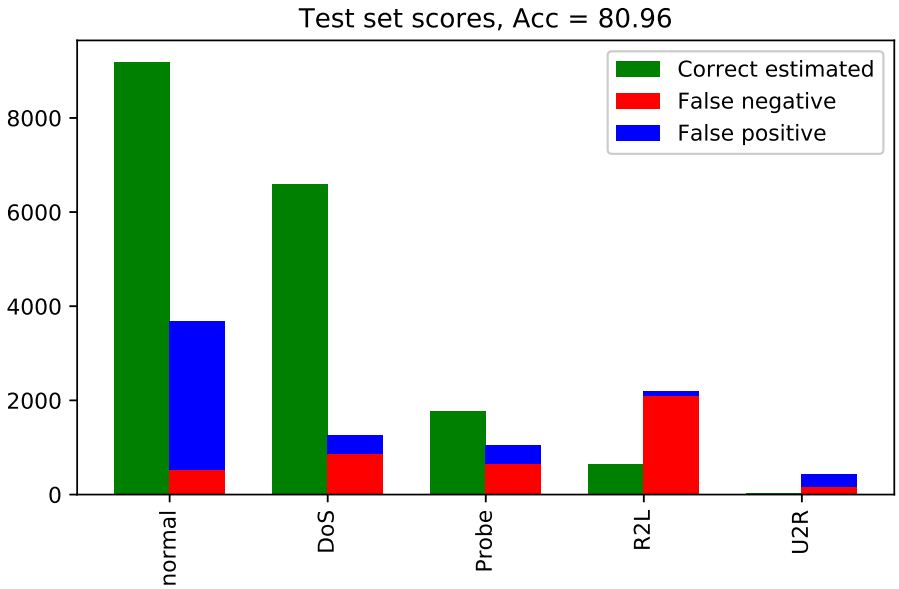

FIGURA 5.11: Test competitivo con mínimo de exploración

Tipo	Estimadas	Correctas	Total	F1 Score
normal	12336	9187	9712	83.34
DoS	6989	6597	7458	91.33
Probe	2155	1770	2421	77.36
R2L	759	658	2753	37.47
U2R	305	39	200	15.45

TABLA 5.4: Resultados del entorno competitivo

5.8. Detector A3C

Como último detector de anomalías, se pretende la implementación un detector con la tecnología descrita en el capítulo 2.7.2. Para ello se necesita modificar completamente el código anterior, ya que la estructuración asíncrona obliga a tener varios trabajadores a requisitos de un coordinador global como se muestra en la figura 2.12.

La intención de este último método consiste en mejorar el algoritmo de *Q-Learning*, no obstante no presentará los beneficios del entorno contrincante del apartado 5.7, por lo que los resultados se verán eclipsados por la distribución de los datos del conjunto de entrenamiento. Pese a ello, este sistema alcanza niveles de detección comparables al entorno competitivo como se puede observar en la figura 5.16 ofreciendo tasas de detección por encima del $80\,\%$.

En este último entorno de aprendizaje se han realizado ciertas modificaciones sobre el código del entorno. En los casos anteriores, para seleccionar un ataque se sigue el siguiente proceso:

1. Se procesa el dataset, tal y como se describe en 3.3
2. Una vez quedan procesados, mezclados y almacenados los datos se determina un índice para comenzar a leer en esa posición de forma aleatoria
3. Se lee de manera secuencial a partir de ese índice en conjuntos de n muestras
4. Cuando se alcanza el final del dataset se comienza desde el principio

En el caso de *A3C*, la forma en la que se seleccionan los datos es similar, pero en vez de definir episodios de n muestras se define un parámetro de fallos. Con este nuevo parámetro se consigue que la longitud de los episodios sea variable ofreciendo mayor visibilidad al aprendizaje. Este parámetro de fallos determina el número de equivocaciones que el detector en el entrenamiento puede cometer. En la figura 5.12 se representa de forma gráfica la evolución del número de episodios alcanzado con el número de fallos fijado. En este caso el número de fallos se corresponde a 10, por lo que el algoritmo es capaz de alcanzar hasta 8000 muestras cometiendo solo 10 fallos. Este valor es únicamente representativo ya que el número de muestras en el entrenamiento no significa que en el conjunto de test obtenga un resultado similar, además como se puede apreciar es un valor muy inestable pero a su vez creciente.

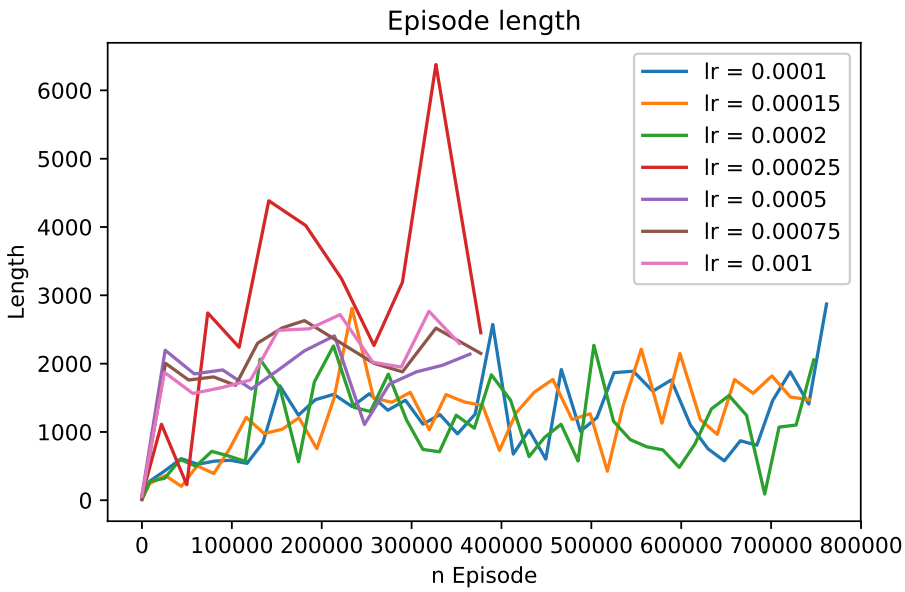

FIGURA 5.12: Longitud de los episodios en el algoritmo A3C

En la figura 5.13 se muestra la evolución del entrenamiento de este algoritmo por medio del número de recompensas obtenidas a lo largo de los episodios. En este ejemplo se muestra la evolución del algoritmo con optimizador **Adam** para distintos valores de la tasa de aprendizaje. Se puede observar claramente que el algoritmo obtiene rápidamente un número elevado de recompensas y que permanece estable con cierta tendencia alcista.

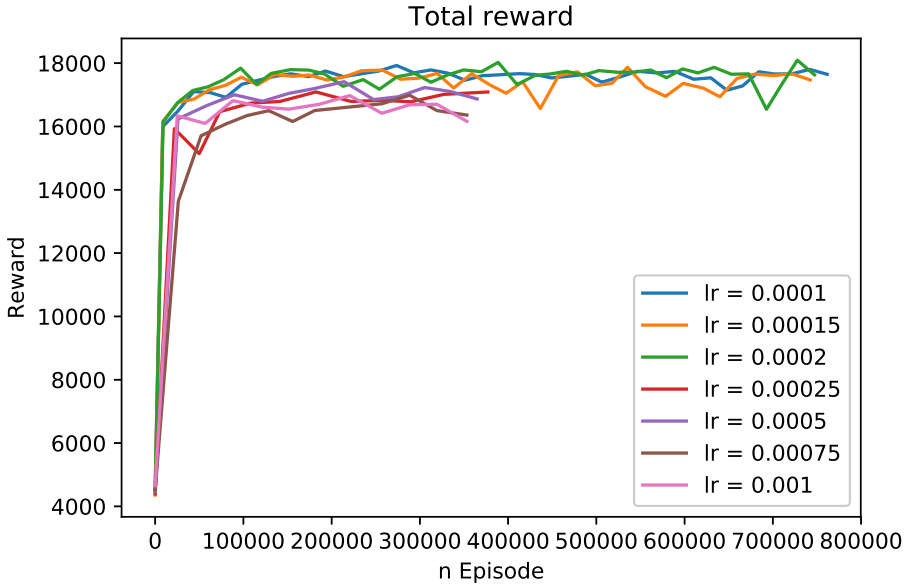

FIGURA 5.13: Número total de recompensas en el entrenamiento del algoritmo A3C

En la figura 5.14 se muestra la precisión en el conjunto de test a lo largo del tiempo para distintos valores de la tasa de aprendizaje. Los valores de la tasa de aprendizaje se han escogido dentro de la misma década ya que ofrece precisiones de calidad. Estas ligeras variaciones se pueden utilizar para escoger el valor óptimo de la tasa de aprendizaje, teniendo en cuenta que este proceso requiere de una alta cantidad de recursos de tiempo o procesamiento.

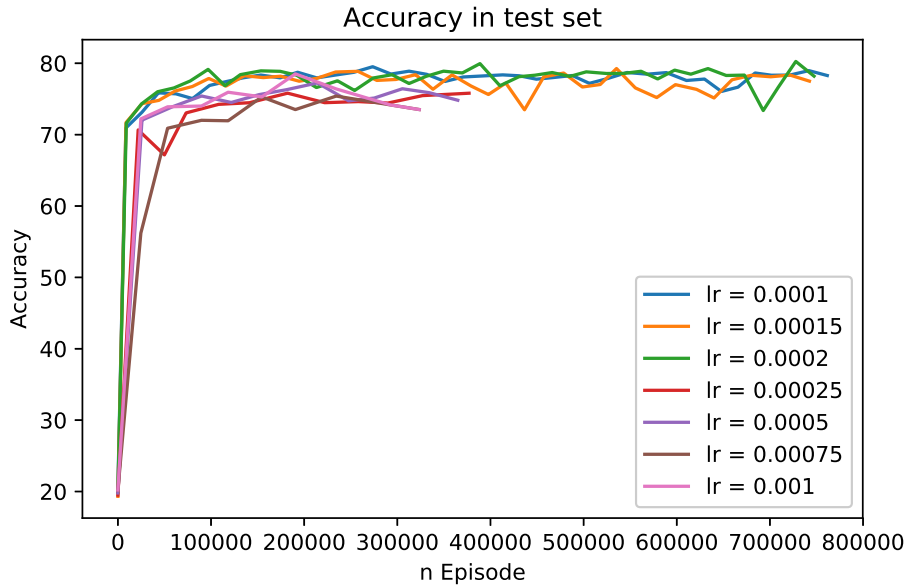

FIGURA 5.14: Precisión en el conjunto de test del algoritmo A3C

En la figura 5.15 se muestra la evolución de la entropía del detector para las distintas tasas de aprendizaje. Para este caso, la entropía dispone de cinco estados posibles por lo que su ecuación es la siguiente:

$$\begin{aligned} H &= -p_1 \log_2(p_1) - p_2 \log_2(p_2) - p_3 \log_2(p_3) - p_4 \log_2(p_4) - p_5 \log_2(p_5) \\ &= -\sum_{i=1}^{5} \log_2(p_i) \end{aligned} \tag{5.4}$$

La implementación de esta ecuación en *tensorflow* requiere únicamente una línea de código y la representación de la figura 5.15 se corresponde con la media de los valores de las entropías de los agentes en cada intervalo:

```
1 entropy = -tf.reduce_sum(probs * tf.log(probs), 1, name="entropy")
2 entropy_mean = tf.reduce_mean(entropy, name="entropy_mean")
```

En el detector *A3C* la entropía representa el dilema de la exploración-explotación, por lo que valores altos de entropía significan altos valores de exploración. Por el contrario, valores bajos de entropía significa que el algoritmo está explotando el conocimiento aprendido. La evolución de estas gráficas muestra un comportamiento inicialmente exploratorio mientras que a lo largo de los episodios termina estabilizándose en un comportamiento de explotación. Para la correcta visualización, la figura 5.15 se

encuentra suavizada, ya que la variación del valor de la entropía es muy alta. Esto implica que por muy bajo que se sitúe la exploración, siempre hay momentos en los que la entropía obtiene un valor elevado y permite una exploración temporal de posibles nuevas acciones. En la misma gráfica, se puede apreciar que todos los valores de la tasa de aprendizaje muestran comportamientos similares, por lo que las gráficas de entropía no ayudan a escoger un valor óptimo de la tasa de aprendizaje, pero dan una idea de la tasa de exploración.

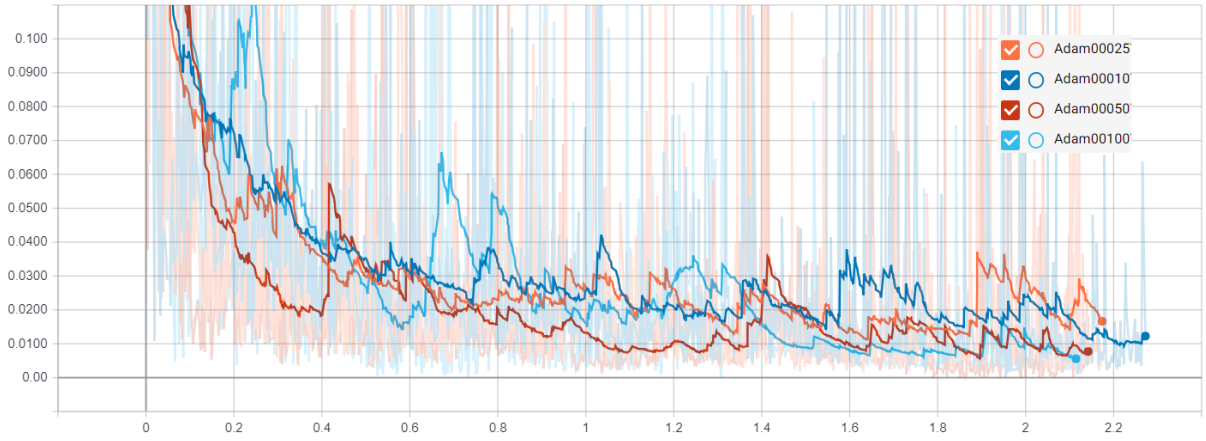

FIGURA 5.15: Entropía de los agentes

La representación de los resultados de este detector se muestra en la figura 5.16. Dependiendo de los hiperparámetros escogidos los resultados mostrarán mayor variación en la clasificación o mayor precisión en las clases más repetidas como la normal, *DoS* o *Probe*. Este comportamiento de centrarse únicamente en las clases más repetidas se explota muy profundamente en los detectores basados únicamente en *Q-learning*. Gracias al entrenamiento asíncrono se extrae la información de mayor cantidad de eventos por lo que A3C ofrece una clasificación correcta a valores menos probables en el conjunto de entrenamiento como *R2L* o *U2R*.

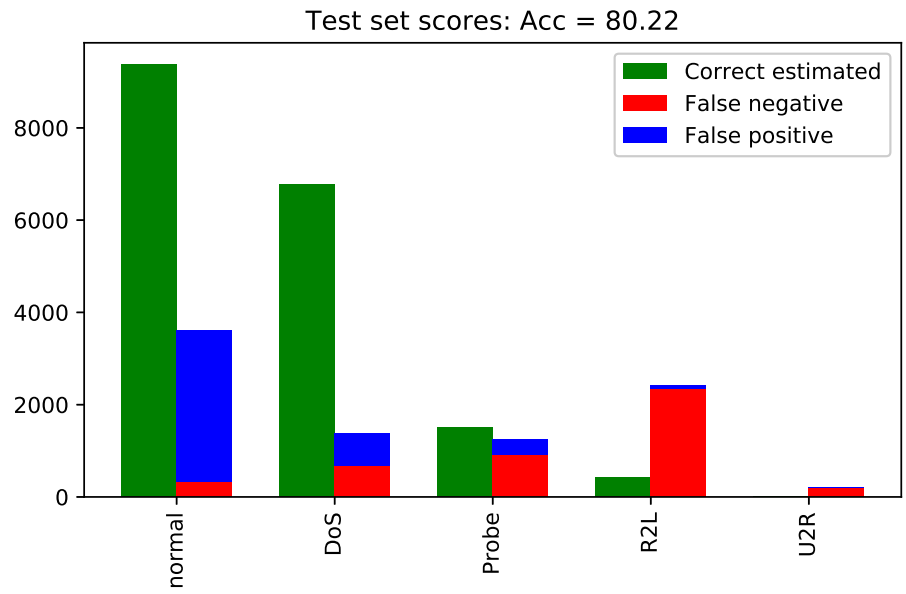

FIGURA 5.16: Test del agente A3C

En la tabla 5.5 se muestran los resultados detallados del conjunto de datos de test. En este caso se aprecia una detección conjunta de entorno al $80\,\%$. Como es de esperar por la distribución de muestras del conjunto de entrenamiento, la mayoría de las muestras correspondientes con ataques *DoS* o normal se clasifican correctamente, mientras que las clases más difíciles como *R2L* o *U2R* obtienen ratios de detección mucho peores pero siendo mejores que en los detectores basados en *Q-learning* y *DDQN*. Estos resultados frente a los correspondientes al entorno competitivo muestran incluso menos falsos positivos para las clases *R2L* y *U2R*, que por otro lado implica también menor tasa de detección. Esto se podría solucionar modificando ligeramente el parámetro de exploración de la entropía β de la ecuación de actualización del gradiente de la política (2.48) o de la función de pérdidas (2.49), pero ocasionaría a su vez un aumento de los falsos positivos.

Tipo	Estimadas	Correctas	Total	F1 Score
normal	12651	9375	9711	83.62
DoS	7476	6781	7458	90.01
Probe	1853	1511	2421	70.71
R2L	482	413	2754	25.52
U2R	22	5	200	4.50

TABLA 5.5: Resultados de A3C

Capítulo 6

Conclusiones

Para finalizar el estudio de los distintos métodos de detección de anomalías utilizando *reinforcement learning* se procede a comparar los distintos algoritmos implementados, las tasas de detección alcanzadas, así como la utilidad de estos posibles detectores.

A continuación se muestran los beneficios y problemas de cada uno de los algoritmos implementados. Uno de los problemas que afecta a todos ellos, es la dificultad para determinar de forma correcta el ataque que ha sucedido. En *reinforcement learning* no se entrena la red neuronal con un entorno supervisado y etiquetado, en su lugar, se utilizan recompensas positivas y/o negativas. Este hecho ocasiona un inconveniente a la hora de acertar en la acción cuando el número de categorías a clasificar es alta. También afecta en el tiempo de entrenamiento porque el algoritmo aprende a base de prueba y error, es decir, cuando una acción es correcta, aprende de ella y además sabe que ha acertado al igual que sabe si ha fallado aunque desconozca la acción correcta. Por otro lado, este problema es un gran beneficio en otros entornos en los que una acción no se puede determinar como buena o mala pero sí que se puede determinar si esta acción repercute en el futuro de forma positiva o negativa. Esto puede ser muy beneficioso a la hora de encontrar no solo acciones sino combinaciones de ellas. Otro problema de estas implementaciones o más bien un inconveniente, es que los algoritmos de *RL* se han diseñado para entornos altamente correlados, por lo que estas características no se pueden explotar en el entorno de detección de anomalías.

6.1. Comparación

Se han estudiado tres tipos de detectores:

- Detector básico: Este detector es útil para decidir si hay anomalía o no. El mayor inconveniente de este detector es que imposibilita la reacción frente a las anomalías. Únicamente se podría cortar el trafico o dejarlo pasar.
- Detector múltiple: Este detector es capaz de determinar el ataque en específico que sucede. Tiene un gran inconveniente y es que es muy complejo entrenar un clasificador con tanta cantidad de clases únicamente a base de prueba y error. Por otro lado, este clasificador es muy útil frente a una reacción para el ataque en específico.
- Detector de tipo: En una posición intermedia a los anteriores se sitúa el detector del tipo de ataque. Para una posible reacción, es suficiente con conocer el tipo de ataque que esta sucediendo para poder poner solución, por lo que este detector es el que se ha estudiado más en profundidad.

Por otro lado, se han estudiado cuatro métodos principales:

- *Q-Learning*: Este ha sido el método básico implementado para la detección, con el se logran rápidamente valores de detección de hasta un $70\,\%$.
- *DDQN*: Con estas mejoras aplicadas al detector simple de *Q-Learning* se consigue mayor estabilidad en el aprendizaje y un incremento notable en la detección para el conjunto de test. Con las mejoras simples de *DDQN* se consigue aumentar la detección en un $5\,\%$ aproximadamente y si además se añade la arquitectura *Dueling* el aumento es aun mayor en torno al $8\,\%$. Además se nota una reducción en la varianza, comparable entre los resultados obtenidos en las figuras 4.1 y 5.2.
- *Multi Agent RL*: El método competitivo supone una mejora notable en cuanto a la aparición de sucesos menos probables. Gracias a esta modificación, la tasa de detección aumenta hasta un $80\,\%$ en el conjunto de test, lo que implica una mayor generalización sobre los datos que no aparecen en el conjunto de entrenamiento. Además, este escenario representa de un modo más natural lo que puede suceder en la realidad.
- *A3C*: El detector con esta tecnología supone un cambio drástico a los anteriores. Gracias a este detector y sin la necesidad de manejar un conjunto de ataques alterado o inteligente, se consiguen detecciones de entorno al $80\,\%$. El uso de múltiples agentes de forma asíncrona asegura que se explora de forma más eficiente el conjunto de datos del entorno.

De entre todos los detectores estudiados, los únicos que son capaces de detectar de forma óptima ataques *R2L* o incluso alguno *U2R*, son el entorno competitivo, el *A3C* y en pequeña medida el *DDQN* con arquitectura *Dueling*. El número de falsos positivos en todos ellos es muy bajo. El mayor problema aparece con los no detectados debido a la aparición de nuevos ataques en el conjunto de test. Los otros detectores se centran en detectar la mayor cantidad de casos normales o *DoS*, por lo que se centran en estas detecciones ocasionando cero detecciones en *R2L* y *U2R*.

6.2. Conclusiones

Los mejores detectores logrados aprovechan los beneficios ya sea del propio entorno competitivo o del aprendizaje de políticas asíncronas que permiten una exploración y aprendizaje más eficiente de las muestras. Teniendo en cuenta esto, se puede decir que la tarea es complicada para los agentes de *reinforcement learning*, ya que no disponen de los beneficios de los que dispone un entorno supervisado de aprendizaje automático. Este problema se soluciona a la hora de explorar nuevas acciones para los casos, lo que origina una alta varianza de los métodos. Dada la falta de correlación entre las muestras y la alta cantidad de clases distintas, el utilizar *RL* en esta tarea de detección no es la mejor opción posible. En otros casos en los que únicamente se disponga del conocimiento de si un ataque ha ocasionado problemas o no a un dispositivo final, el aplicar métodos de *RL* para aprender a reaccionar de la mejor forma posible ante estos problemas puede ser una opción muy interesante. Los métodos de *RL* son la mejor opción cuando el aprendizaje del agente se mueve en un entorno completamente desconocido e independiente a él. En este caso, un agente de *RL* se puede beneficiar ya

que no necesita de la supervisión de una persona, por lo que con unas simples métricas de qué es bueno y qué es malo puede aprender a reaccionar ante cualquier situación nueva. Este tipo de métodos serán muy utilizados cuando la alta cantidad de ataques distintos hagan inviable la creación de algoritmos supervisados.

La calidad de los detectores implementados depende directamente de las muestras que se utilizan para su entrenamiento. Por ello, como se ha explicado con anterioridad, se acaba utilizando el dataset **NSL-KDD**. Este conjunto de datos dispone de 23 tipos de ataques distintos en el conjunto de entrenamiento como se muestra en la figura 6.1. De esos ataques se puede mapear la distribución de los tipos como se muestra en la figura 6.2. Esta distribución de ataques es muy similar a las distribuciones de estimaciones de todos los métodos. La mayor parte de las clasificaciones se asignan a la clase normal, después a la *DoS* y como tercera posición *Probe*. Partiendo de esta distribución, el conjunto de muestras *U2R* es sumamente bajo y por lo tanto es lógico que en el conjunto de test no se detecten de forma correcta. Por mucho que se intente entrenar el clasificador con este conjunto de datos, se va a conseguir un sesgo hacia las clases más populares debido a la dificultad de que asigne de forma correcta las muestras poco frecuentes. Si de un total de 125973 muestras de entrenamiento, solo 52 son *U2R* y además de esto, el algoritmo tiene que adivinar por medio de prueba y error la clase correcta, se puede prever que no va a funcionar de un modo correcto.

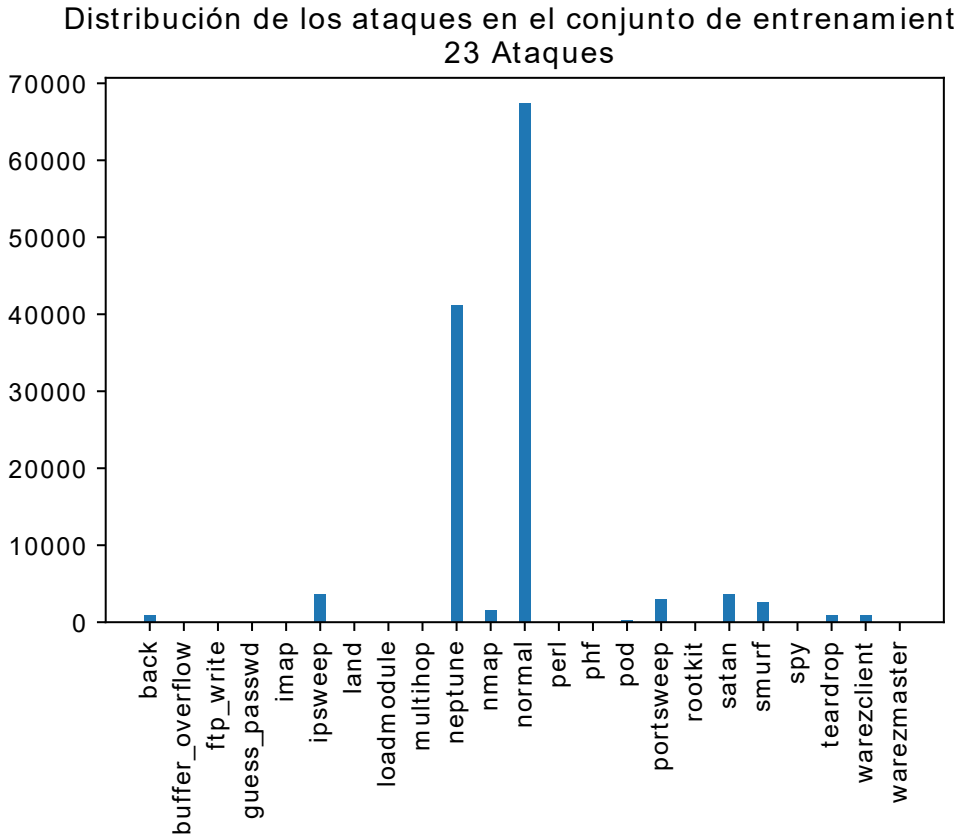

FIGURA 6.1: Distribución de los ataques en el conjunto de entrenamiento

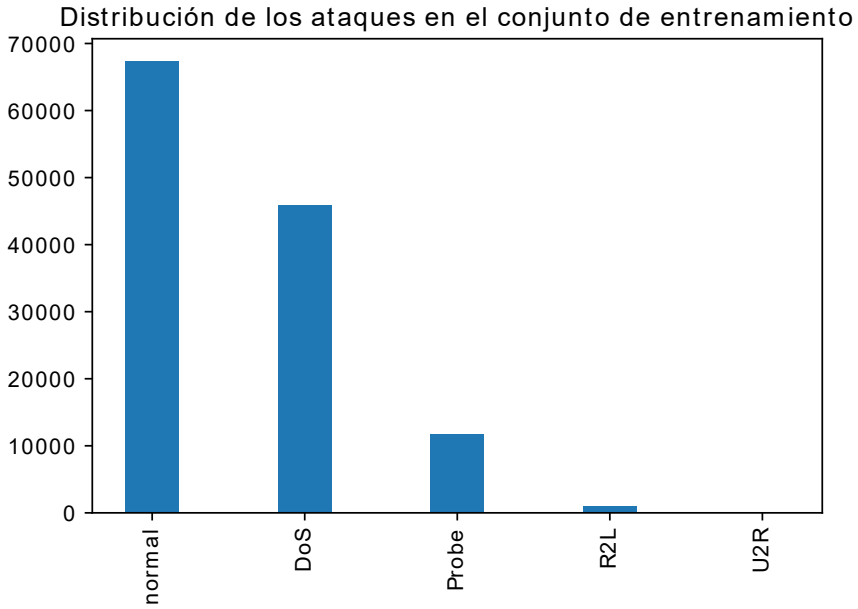

FIGURA 6.2: Distribución de los tipos de ataques en el conjunto de entrenamiento

Por otro lado, el estudio del conjunto de datos de test puede dar información adicional del comportamiento de los detectores. En la figura 6.3 se muestra la distribución de los ataques para el conjunto de datos de test. En esta figura se puede ver que las clases se siguen centrando en la normal y como segunda clase *neptune* al igual que pasa en el conjunto de entrenamiento. En este caso, el número de ataques ya no es de 23 sino que ahora son 38 lo que añade todavía más complejidad. Si a este conjunto se le mapea por medio de su tipo de ataque, se obtiene la representación de la figura 6.4. Esta figura muestra una distribución bastante distinta a la del conjunto de entrenamiento. Por un lado, las clases normal y *DoS* siguen siendo las más repetidas, pero por otro lado ahora existen muchos más casos de *R2L* que en el conjunto de entrenamiento. Este aumento en la clase *R2L* ocasiona una falta de detecciones notable ya que no se dispone de suficiente información extraíble en el conjunto de entrenamiento. El resultado de esta variación ocasiona que en la mayor parte de los detectores, la clase correspondiente a *R2L* tenga la mayor parte de estimaciones incorrectas (falsos negativos). Por otro lado, estas muestras se clasifican de forma errónea normalmente en la clase normal lo que podría ocasionar problemas importantes de seguridad (falsos positivos).

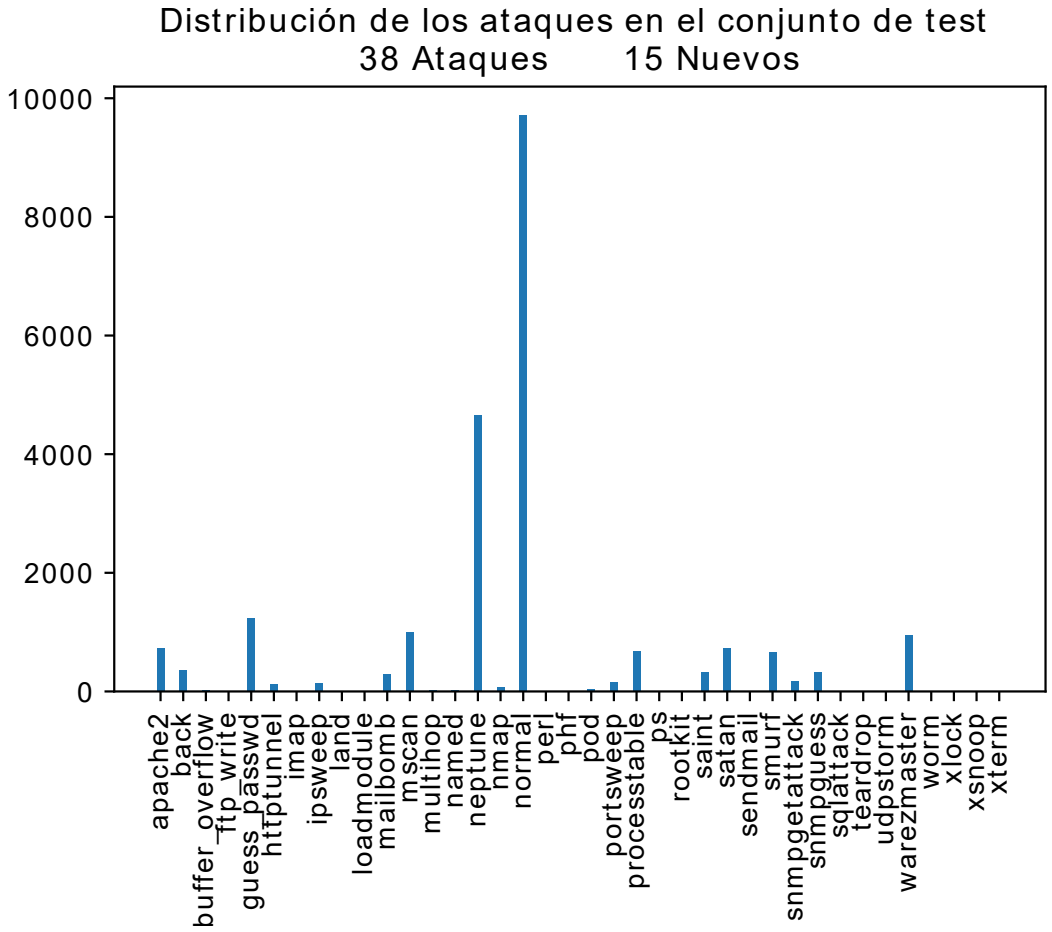

FIGURA 6.3: Distribución de los ataques en el conjunto de test

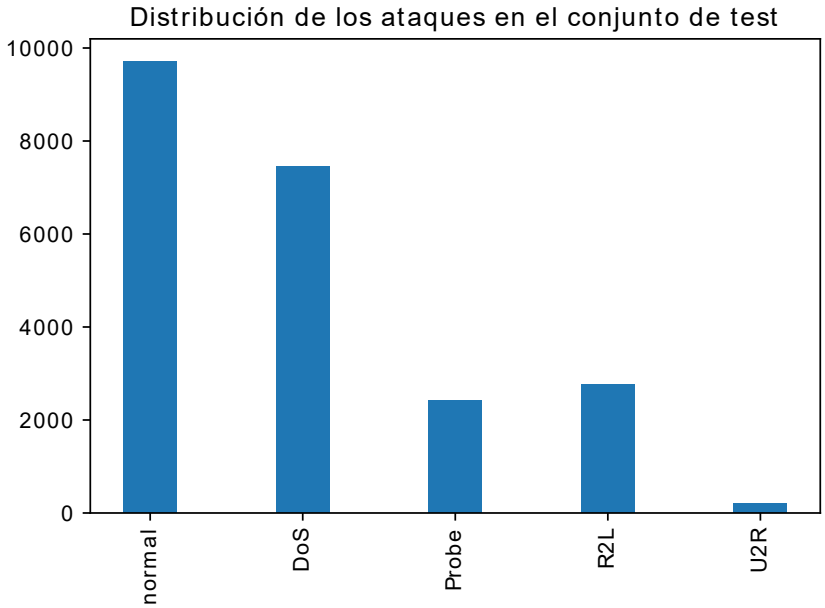

FIGURA 6.4: Distribución de los tipos de ataques en el conjunto de test

Vistas las distribuciones de datos de los conjuntos de entrenamiento y test, se comprende de mejor forma las equivocaciones que sufren los detectores. Sin ningún tipo de modificación en los datos de entrada, la probabilidad de acertar por medio de prueba y error las clases menos repetidas es muy baja y por ello el entorno competitivo muestra tanta mejora a la hora de clasificar los ataques *R2L* gracias al aumento de la frecuencia de estos datos.

Para finalizar, el aspecto fundamental que se puede obtener de este estudio es; que el entrenamiento de clasificadores por medio de prueba y error no es del todo eficiente cuando no se dispone de datos eficientemente distribuidos. Los errores que va a cometer un clasificador que no dispone de suficientes muestras de cada clase por medio de prueba y error van a ser altos incluso en el conjunto de datos de entrenamiento. Esto se podría solucionar con un entrenamiento más largo y repitiendo más veces las muestras menos frecuentes, lo que a su vez ocasionaría un problema de sobreajuste completamente indeseado.

6.3. Líneas futuras y posibles mejoras

Dados los recientes descubrimientos en la teoría de *Reinforcement Learning* las posibles mejoras a estos detectores implican actualizar los modelos con cierta periodicidad para aprovechar los últimos descubrimientos. También queda pendiente la posible aplicación del detector *A3C* utilizando un entorno inteligente y competitivo ya sea por medio de otro agente *A3C* o un simple *Q-Learning*.

Otra de las posibilidades que se plantean, es la posible generación de muestras completamente nuevas a los conjuntos de entrenamiento y test. La generación de muestras se puede lograr utilizando redes generativas antagónicas o adversarias *GAN*. Esto ocasiona un problema y es que los nuevos casos no están clasificados y es posible que incluso no signifiquen nada, pero sería una forma de obtener mecanismos evolutivos para los propios ataques.

Uno de los aspectos fundamentales que se pueden mejorar en estos detectores es la optimización de los hiperparámetros de las redes implementadas. Siempre es posible obtener mejores valores utilizando combinaciones de todos ellos, pero actualmente dada la situación en la que se plantea el proyecto, es imposible computacionalmente.

Otro de los aspectos fundamentales en un detector que se puede modificar es la ingeniería de características o *features*. A las *features* básicas se pueden añadir combinaciones de ellas, suprimir algunas que no aporten información relevante o incluso generar nuevas. Los métodos de reducción de la dimensionalidad son muy utilizados en estos casos consiguiendo mejorar la tasa de detección y el tiempo de procesamiento.

Por último, queda pendiente la posible implementación del entorno adaptado para el dataset *NSL* con la finalidad de que se pueda añadir al entorno de *RL* **OpenAI Gym**. Si se dispone de un entorno adaptado con las mismas interfaces, se pueden comprobar otros detectores diseñados con otros propósitos de forma general.

Bibliografía

[1] DeepMind. *AlphaGo Zero: Learning from scratch*. 2017. URL: https://deepmind.com/blog/alphago-zero-learning-scratch/.

[2] Richard S Sutton y Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[3] David Silver. *UCL Course on RL. Lecture 1: Introduction to Reinforcement Learning*. 2015. URL: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/intro_RL.pdf.

[4] Marco Wiering y Martijn Van Otterlo. «Reinforcement learning». En: *Adaptation, learning, and optimization* 12 (2012).

[5] Csaba Szepesvári. «Algorithms for reinforcement learning». En: *Synthesis lectures on artificial intelligence and machine learning* 4.1 (2010), págs. 1-103.

[6] David Silver. *UCL Course on RL. Lecture 6: Value Function Approximation*. 2015. URL: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/FA.pdf.

[7] Volodymyr Mnih y col. «Human-level control through deep reinforcement learning». En: *Nature* 518.7540 (feb. de 2015), págs. 529-533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[8] Peter Auer. «Using confidence bounds for exploitation-exploration trade-offs». En: *Journal of Machine Learning Research* 3.Nov (2002), págs. 397-422.

[9] Georgios Theocharous y col. «Posterior Sampling for Large Scale Reinforcement Learning». En: *CoRR* abs/1711.07979 (2017). arXiv: 1711.07979. URL: http://arxiv.org/abs/1711.07979.

[10] Daniel Fink. «A compendium of conjugate priors». En: *See http://www. people. cornell. edu/pages/df36/CONJINTRnew % 20TEX. pdf* 46 (1997).

[11] Cameron Davidson-Pilon. *Multi-Armed Bandits*. 2017. URL: https://dataorigami.net/blogs/napkin-folding/79031811-multi-armed-bandits.

[12] David Silver. *UCL Course on RL. Lecture 7: Policy Gradient*. 2015. URL: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/pg.pdf.

[13] Timothy P. Lillicrap y col. «Continuous control with deep reinforcement learning». En: *CoRR* abs/1509.02971 (2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[14] Arthur Juliani. *Simple Reinforcement Learning with Tensorflow. Part 8: Asynchronous Actor-Critic Agents (A3C)*. 2016. URL: https://medium.com/emergent-future.

[15] Volodymyr Mnih y col. «Asynchronous Methods for Deep Reinforcement Learning». En: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[16] M. Tavallaee y col. «A detailed analysis of the KDD CUP 99 data set». En: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. 2009, págs. 1-6. DOI: 10.1109/CISDA.2009.5356528.

[17] Greg Brockman y col. «Openai gym». En: *arXiv preprint arXiv:1606.01540* (2016).

[18] Mahbod Tavallaee y col. «A detailed analysis of the KDD CUP 99 data set». En: *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*. IEEE. 2009, págs. 1-6.

[19] S Revathi y A Malathi. «A detailed analysis on NSL-KDD dataset using various machine learning techniques for intrusion detection». En: *International Journal of Engineering Research and Technology. ESRSA Publications* (2013).

[20] Hado van Hasselt, Arthur Guez y David Silver. «Deep Reinforcement Learning with Double Q-learning». En: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[21] OpenAI. *OpenAI Baselines: DQN*. 2017. URL: https://blog.openai.com/openai-baselines-dqn/.

[22] Ziyu Wang, Nando de Freitas y Marc Lanctot. «Dueling Network Architectures for Deep Reinforcement Learning». En: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.