ETSIT

Escuela Técnica Superior de Ingenieros de Telecomunicación

Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Enrutamiento y Establecimiento Dinámico de Conexiones en Redes de Transporte mediante Aprendizaje por Refuerzo

Autor:

D. Mieszko Ferens

Tutores:

D. Ignacio de Miguel Jiménez

D. Ramón José Durán Barroso

VALLADOLID, SEPTIEMBRE 2020

TRABAJO FIN DE GRADO

TÍTULO:	**Enrutamiento y Establecimiento Dinámico de Conexiones en Redes de Transporte mediante Aprendizaje por Refuerzo**
AUTOR:	**D. Mieszko Ferens**
TUTORES:	**D. Ignacio de Miguel Jiménez** **D. Ramón J. Durán Barroso**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**D. Ignacio de Miguel Jiménez**
VOCAL:	**D. Ramón J. Durán Barroso**
SECRETARIO:	**Dña. Noemí Merayo Álvarez**
SUPLENTE:	**D. Juan Carlos Aguado**
SUPLENTE:	**D. Evaristo J. Abril Domingo**

Resumen

El aprendizaje automático está siendo utilizado en cada vez más ámbitos como una solución a problemas existentes más eficiente que las técnicas tradicionales. Un problema típico en la actualidad es el enrutamiento en redes telemáticas. Relacionado con esta cuestión, es muy importante realizar una asignación eficiente de recursos a la hora de establecer conexiones en estas redes, por lo que recientemente muchos estudios se han centrado en buscar algoritmos de aprendizaje automático que resuelvan esto. En este Trabajo Fin de Grado (TFG) nos centramos en el aprendizaje por refuerzo, una rama del aprendizaje automático en auge. Explicamos sus fundamentos, así como algunos algoritmos para después aplicarlos a dos problemas concretos. El primero es la búsqueda de la ruta de menor coste en una red de comunicaciones, un caso de estudio típico, que empleamos como iniciación al ámbito del aprendizaje por refuerzo. Después nos centramos en el establecimiento dinámico de conexiones en una red de transporte, mostrando el rendimiento en distintos casos para estos algoritmos.

Palabras Clave

Aprendizaje por refuerzo, agente, Deep Q-learning, ChainerRL, OpenAI Gym, enrutamiento, asignación de recursos, redes de transporte

Abstract

Machine learning is being used in an increasing range of applications, as a more efficient solution to existing problems than traditional methods. A common problem in recent times is routing in telematic networks. Related to this issue, efficient resource assignment when establishing connections in these networks is a key issue. Therefore, many recent studies have focused on devising machine learning algorithms to solve these problems. In this Bachelor Thesis we will focus on reinforcement learning, a booming machine learning branch. We explain its fundamentals, as well as several algorithms which we then apply to two specific problems. The first one is the search for the lowest cost route in a communications network, a typical case study, which we use as an initiation into reinforcement learning. Afterwards, we turn our attention to dynamic connection establishment in a transport network, showing the performance of these algorithms in various cases.

Keywords

Reinforcement Learning, agent, Deep Q-learning, ChainerRL, OpenAI Gym, routing, resource assignment, transport networks

Agradecimientos

Deseo realizar un especial agradecimiento a D. Ignacio de Miguel Jiménez por dedicar tiempo a iniciarme en el ámbito del aprendizaje automático, y especialmente, el aprendizaje por refuerzo de cara a realizar este TFG. También quiero agradecer su tutorización a lo largo de la realización de la memoria, cuya realimentación y revisiones fueron muy instructivas para mejorar mucho la calidad del documento.

También quiero dedicar una mención al Dpto. de Teoría de la Señal y Comunicaciones e Ingeniería Telemática por la oportunidad de realizar una beca de colaboración con ellos, cosa que ha permitido mayor dedicación al trabajo y ha supuesto una profundización mayor en el ámbito, y a D. Ramón J. Durán Barroso por su realimentación al proyecto.

La investigación desarrollada en este Trabajo Fin de Grado ha sido financiada por el Ministerio de Ciencia, Innovación y Universidades en el marco del proyecto ONOFRE-2 (TEC2017-84423-C3-1- P) y la red de investigación Go2Edge (RED2018-102585-T), por la Consejería de Educación de la Junta de Castilla y León en el marco del proyecto ROBIN (VA085G19), y por el Fondo Europeo de Desarrollo Regional FEDER a través del proyecto DISRUPTIVE del Programa Interreg V-A España-Portugal (POCTEP) 2014-2020. Las opiniones son de exclusiva responsabilidad del autor que las emite.

Índice

1. Introducción ... 9
1.1. Motivación ... 9
1.2. Objetivos ... 10
1.3. Estructura de la memoria ... 10
2. Fundamentos de aprendizaje por refuerzo, librerías y algoritmos ... 11
2.1. Aprendizaje por refuerzo ... 11
2.2. Herramientas ... 13
2.2.1. Comparación de librerías de RL ... 13
2.2.2. Algoritmos de aprendizaje por refuerzo empleados ... 15
2.2.2.1. DDQN (Double Deep Q-Network) ... 16
2.2.2.2. SARSA (State-Action-Reward-State-Action) ... 16
2.2.2.3. PAL (Persistent Advantage Learning) ... 17
2.2.2.4. TRPO (Trust Region Policy Optimization) ... 17
3. Búsqueda de la ruta de menor coste ... 18
3.1. Alternativas para la búsqueda de la ruta de menor coste ... 18
3.2. Agentes RL ... 18
3.3. Entorno ... 19
3.3.1. Ideas generales ... 19
3.3.2. Observación ... 19
3.3.3. Acciones ... 20
3.3.4. Recompensa ... 21
3.4. Experimentos ... 21
3.4.1. Experimento en red con bifurcación ... 22
3.4.2. Experimento en red con sumidero ... 28
3.5. Conclusiones ... 33
4. Establecimiento dinámico de conexiones en redes de transporte ... 34
4.1. Entorno ... 34
4.1.1. Ideas generales ... 34
4.1.2. Observación y recompensa ... 36
4.1.3. Fuente de tráfico ... 37
4.2. Agentes RL ... 38
4.3. Implementación de algoritmos tradicionales usados como comparativa ... 39
4.4. Experimentos ... 40

4.4.1. Circuitos transportando tráfico homogéneo con cargas iguales y con carga de red distribuida uniformemente 40

4.4.2. Circuitos transportando tráfico heterogéneo y con carga de red distribuida uniformemente 42

4.4.3. Circuitos transportando tráfico heterogéneo, con carga de red distribuida uniformemente y control de admisión 46

4.4.4. Circuitos transportando tráfico homogéneo, con carga de red distribuida uniformemente y control de admisión 49

4.5. Análisis del entrenamiento propuesto y posibles mejoras 51

4.6. Conclusiones 54

5. Conclusiones y líneas futuras 55

5.1. Conclusiones 55

5.2. Líneas futuras 56

Bibliografía 58

1. Introducción

1.1. Motivación

No hace falta demostrar la importancia de las redes de telecomunicaciones hoy en día, fundamentales para todo tipo de aplicaciones, que ven un uso muy extenso y que son vitales para el correcto funcionamiento de diversos servicios. Un problema al que se enfrentan estas redes constantemente es cómo repartir el uso de los recursos de los que disponen para alcanzar el mejor rendimiento posible.

Tradicionalmente esto se aborda con algoritmos como los basados en la ruta más corta. Su filosofía es sencilla, cuantos menos nodos y enlaces se utilicen en una conexión, menor será el consumo de recursos y, en general (aunque no necesariamente), menor serán los retardos experimentados. El problema que viene al usar estas soluciones es que no tienen por qué ser óptimas. Una asignación más compleja de los recursos puede ser necesaria para reducir las cargas en puntos concretos de una red y así permitir un uso más eficiente de esta. De poco sirve tener la mayoría de los enlaces disponibles si unos pocos enlaces clave se congestionan y las situaciones de bloqueo se repiten de forma reiterada.

Hay muchas formas de intentar corregir este último problema, pero en muchos casos no son sencillas de poner en práctica debido a limitaciones que se presentan en redes reales como puede ser la falta de información sobre el estado exacto de toda la red en cada momento. Obviamente esta falta de información supone que puede llegar a ser imposible calcular con precisión el reparto de recursos adecuado en cada momento, por lo que esto deberá ser estimado. De hecho, esto es uno de los puntos fuertes de un algoritmo como el de la ruta más corta para este tipo de aplicaciones. Solamente necesita saber la ruta más corta desde el origen hasta el destino, y para esto ni siquiera hace falta conocerla en todos los nodos de la red. Bastaría con que cada nodo supiera cual es el enlace de salida que lleva al destino por la ruta más corta.

Viendo lo anterior entramos ahora en el planteamiento de una solución que pueda otorgar todas estas ventajas y con potencial para proporcionar resultados mejores que los de los métodos tradicionales. Se trata del aprendizaje automático, que ha visto un uso muy extenso en multitud de aplicaciones a día de hoy. Una de sus ramas, el aprendizaje por refuerzo (*Reinforcement Learning*), es muy utilizado cuando se requiere un agente que sea capaz de realizar un control de nivel humano en un sistema. Un tal sistema podría ser un controlador de enrutamiento de conexiones a través de una red telemática. Pues bien, en esta memoria nos proponemos probar cómo de aplicable es realmente el aprendizaje por refuerzo al enrutamiento en redes y evaluar los retos que supone.

Debido a que no queremos plantear un sistema que sea inviable en redes reales, procuraremos limitar todo lo posible la información que recibirán estos algoritmos mientras entrenan para que así se puedan adaptar más fácilmente a estos casos. Recordemos que no tiene sentido una solución que necesite para su funcionamiento información que luego no será obtenible en la práctica. Aquí introduciremos el artículo por J. Suárez-Varela *et al.* [1], en el que los autores exploran un problema muy similar a lo que nosotros nos proponemos. En su caso, intentan que agentes realicen un uso lo más eficiente posible de los recursos de una red, estableciendo conexiones que perduran para siempre hasta que haya una situación de bloqueo. Este es un estudio interesante debido a que muestra la capacidad de estos algoritmos para aprender a adaptarse a una topología de red, así como sus características, para así poder hacer un buen uso de sus recursos. Sin embargo, nosotros queremos un escenario dinámico en lugar de incremental, por lo que buscaremos

que un agente establezca conexiones que no perduran para siempre en una red de forma continua.

1.2. Objetivos

En esta memoria nos proponemos obtener resultados favorables para justificar el uso del aprendizaje por refuerzo en redes telemáticas explorando el potencial de estos algoritmos en un caso del enrutamiento y especialmente en el establecimiento dinámico de conexiones. A continuación, presentamos la lista de objetivos del TFG:

- Comparar distintas librerías de algoritmos de RL, y seleccionar una de ellas.
- Resolver el problema de cálculo de la ruta de menor coste en una red mediante técnicas de aprendizaje por refuerzo profundo.
- Resolver el problema de establecimiento dinámico de conexiones en una red real (NSFNet) mediante técnicas de aprendizaje por refuerzo profundo.
- Evaluar el impacto de distintos algoritmos de RL y sus parámetros asociados en los resultados obtenidos.
- Analizar distintos escenarios de enrutamiento dinámico de conexiones en una red real (NSFNet) para discernir las ventajas del RL en cada uno.

1.3. Estructura de la memoria

Estructuramos la memoria por orden de complejidad. Esto quiere decir que inicialmente en el capítulo 2 introducimos el aprendizaje por refuerzo y las herramientas que emplearemos en el resto de la memoria, así como algunos conceptos fundamentales que se verán de forma repetida en los siguientes capítulos.

Después nos encontraremos con el capítulo 3, en el que probamos hacer uso de las herramientas descritas para resolver con aprendizaje por refuerzo profundo un problema bien conocido, la búsqueda de la ruta más corta (en realidad, la ruta de menor coste). Este capítulo sirve fundamentalmente para familiarizarse con las herramientas seleccionadas y plantea un problema más sencillo que el del objetivo final.

A continuación, tenemos el capítulo 4, en el que abordamos el problema fundamental del trabajo. Aquí buscamos probar la utilidad de los algoritmos de aprendizaje por refuerzo en enrutamiento en redes telemáticas, para un escenario dinámico de establecimiento y liberación de conexiones. Si bien no mostramos el óptimo absoluto que son capaces de lograr estos algoritmos, demostramos su potencial, explorando diversos aspectos de interés presentes en el problema y cómo lidian con ello nuestras soluciones.

Por último, en el capítulo 5 exponemos las principales conclusiones extraídas y posibles líneas de trabajo futuras.

2. Fundamentos de aprendizaje por refuerzo, librerías y algoritmos

2.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo es una de las ramas prevalentes del ámbito del aprendizaje automático hoy en día. En esta sección haremos una breve explicación de qué es basándonos en [2].

Cuando hablamos de aprendizaje por refuerzo (*Reinforcement Learning*, RL) nos referimos a uno o varios agentes interactuando con un entorno. Un agente toma una decisión que implica la realización de una acción (aunque la acción puede ser no hacer nada), y esto conlleva un resultado en el entorno. Este resultado es visible para el agente en forma de observación y de recompensa inmediata. A medida que un agente va tomando acciones, recibirá a su vez el resultado de dichas acciones. Su objetivo es entonces obtener el mejor resultado posible. Llegar al mejor resultado es análogo a que el agente maximice la recompensa acumulada a largo plazo.

Como ya hemos mencionado, a una observación la seguirá una acción de agente, lo que conllevará una recompensa. Este ciclo se repite dando lugar a la historia, que es la secuencia de observaciones, acciones y recompensas. Sabiendo esto podemos definir el sistema como una máquina de estados, cuyos estados son una función de la historia, es decir, el estado puede verse como un resumen de la historia del sistema. Estando en un cierto estado, al tomar una cierta acción se pasa a un nuevo estado. Estos cambios de estado tendrán vinculados las recompensas que los acompañan, por lo que, conociendo todos los estados del sistema, la probabilidad de pasar a otro estado en función de la acción tomada, y las recompensas asociadas, el agente puede tomar las acciones óptimas en cada estado.

El problema típico al que nos enfrentamos es que normalmente se desconoce al menos parte de la información del entorno. Esto implica que los agentes necesitan descubrir los estados, el impacto de sus acciones para ir moviéndose entre estados y las recompensas. Para ello se explora, es decir, un agente no siempre tomará la acción que considere óptima para poder descubrir posibles resultados que aún desconoce. Básicamente los agentes deben balancear la explotación del entorno (toma de las acciones óptimas conocidas) con la exploración (toma de acciones no necesariamente óptimas). La razón por la que es importante balancear ambas cosas es que queremos que el agente aprenda las acciones óptimas en cada estado y para ello necesita explorar lo que ocurre al tomar distintas acciones, y por tanto tiene que probar "cosas nuevas" de vez en cuando, aunque prioritariamente ejecute aquellas acciones que considera óptimas hasta ese momento para explotar los beneficios de las mismas, esto es, para aprovecharse de la recompensa (a largo plazo) que llevan asociadas. Existen distintas alternativas para realizar la exploración y, dependiendo del caso, pueden ser más convenientes unas u otras.

Debido a que la toma de acciones subóptimas en los primeros estados puede conllevar la llegada a estados futuros que no otorgan buenos resultados sin importar las decisiones del agente, es importante que este tome acciones óptimas alternadas con las exploratorias para centrarse en los casos más prometedores. En sistemas complejos, como no es razonable alcanzar todos los posibles estados para observar sus resultados, es importante que el agente generalice. Esto quiere decir que debería ser capaz de tomar la acción óptima en un estado que nunca ha visto antes. Al fin y al cabo, un agente basará su acción en un punto de

la historia determinado en la última observación recibida, y en su experiencia previa (observaciones y recompensas previas). Recordemos que el objetivo principal es maximizar la recompensa acumulada a largo plazo, no la inmediata.

Entonces, ¿qué define las acciones que tomará un agente en un momento determinado? La política. La política es una función que indica la acción a tomar en cada estado (política determinista) o, de forma más general, una función que indica la probabilidad de tomar una acción concreta cuando se está en un determinado estado (política estocástica). La acción mapeada a cada estado se actualiza a medida que el agente toma acciones y recibe nuevas observaciones y recompensas. Volvemos a que es necesaria la toma de acciones posiblemente subóptimas para poder descubrir mejores recompensas acumuladas a largo plazo.

Aquí se deben distinguir dos tipos fundamentales de algoritmos de RL, los *on-policy* y los *off-policy*. La diferencia fundamental es qué política se evalúa o mejora (*target policy*) y qué política se sigue para actuar y obtener los datos (*behaviour policy*). En el caso de los métodos *on-policy*, ambas *target* y *behaviour policies* son la misma. Esto quiere decir que el agente mejora la misma política que sigue en la toma de sus acciones. La fundamental característica de este método es que durante la evaluación se tiene en cuenta la exploración incluida en la política, lo que hace que la política que se alcanza no sea verdaderamente óptima (en un escenario en el que ya no se realizara exploración). Por otra parte, para los métodos *off-policy* esto no es así. Como la *target policy* y la *behaviour policy* no son las mismas, la política que usa durante la evaluación no es la misma que la que se usa para tomar acciones. Esto implica que durante la evaluación no se tendrá en cuenta ninguna exploración, sino simplemente la mejor acción posible desde el estado correspondiente, alcanzándose así (potencialmente) la política óptima.

Asociado a la política tenemos el concepto de función de valor. Esta función predice la recompensa media acumulada (a largo plazo) desde un estado (o desde que se toma una acción concreta) tomando a partir de ese momento las acciones dictadas por la política. Básicamente es una métrica de lo bueno que es estar en un estado (función de valor del estado) o tomar una acción concreta en un estado (función de valor de la acción). Las funciones de valor son, por tanto, útiles para evaluar el funcionamiento de una política concreta y para determinar cuál es la política óptima de un sistema.

Acabamos de mencionar el concepto de recompensa acumulada. A esto lo llamaremos también retorno. En su cálculo se tiene en cuenta la recompensa inmediata, pero también las recompensas que se irán recibiendo en los instantes posteriores. Aquí conviene mencionar también el factor de descuento (γ), que indica el peso que se les da a las recompensas posteriores. Esto se incluye porque normalmente es interesante dar menor importancia a las recompensas posteriores, aunque es posible tener todas en cuenta por igual. También es posible fijar $\gamma = 0$, lo cual se traduce en considerar únicamente la recompensa inmediata. A esto último se le llama tener un sistema miope.

Aquí conviene diferenciar entre tareas continuas y tareas episódicas. En las tareas a resolver mediante aprendizaje por refuerzo puede haber un estado final (tarea episódica) o no (tarea continua). En caso de no tenerlo será importante ajustar correctamente el valor del factor de descuento (pues en el caso de ser la unidad, el retorno podría ser infinito al estar el agente moviéndose "eternamente" por el sistema y recogiendo recompensas).

Para calcular las funciones de valor, se usan las denominadas ecuaciones de Bellman. Estas ecuaciones relacionan la función de valor de un estado (o acción) con las funciones de valor

de los estados (o acciones) inmediatamente siguientes. Aquí obviamente entra en juego el factor de descuento. Dependiendo de si se conoce el modelo del sistema (esto es, si se conoce la probabilidad de pasar al estado s' y de recibir la recompensa inmediata r, cuando estamos en el estado s y tomamos la acción a, para todos los estados y acciones posibles) o si no se conoce, y dependiendo de si lo que se desea es evaluar una política o encontrar la política óptima, así se usan distintas técnicas/algoritmos para resolver esas ecuaciones o apoyarse en ellas para obtener las funciones de valor.

Como el objetivo de esta memoria no es describir todos los avances en RL hasta hoy en día, nos centraremos en los algoritmos que nos interesan. En concreto, hablaremos de la librería ChainerRL [3] y las utilidades que ofrece.

2.2. Herramientas

2.2.1. Comparación de librerías de RL

En este TFG nos hemos decantado por el uso de la librería ChainerRL [3]. Antes de comenzar, comentemos las razones para usar, en nuestro caso, esta librería. La decisión fue tomada debido a una breve comparación de varias posibilidades, las cuales presentamos en la Tabla 1.

	Documentación	**Visualizador**	**Número de algoritmos que implementa**
ChainerRL [3]	Buena	Tiene	Muchos
KerasRL [4]	Buena	Tiene	Muchos
Tensorforce [5]	Buena	Tiene	Muchos
MushroomRL [6]	Aceptable	No tiene	Pocos
Pyqlearning [7]	Aceptable	No tiene	Pocos
RL_coach [8]	Escasa	No tiene	Pocos
NAME_RL [9]	Buena	No tiene	Pocos

Tabla 1 – Comparación entre librerías de aprendizaje por refuerzo

El objetivo de esta comparación no era decidir cuál de estas librerías es mejor, sino cuál era la que usaríamos para nuestros experimentos. Lo que buscábamos era una librería que permitiera implementar agentes de RL de forma sencilla y sin que fuera necesario tener conocimientos avanzados sobre estos. Esto implica que queríamos también muchos ejemplos de uso, así como una buena documentación que lo explicara. Si nos fijamos en la Tabla 1, de las siete librerías destacan ChainerRL [3], KerasRL [4] y Tensorforce [5] debido a que sus funcionalidades y APIs son muy buenas. Las otras cuatro [6] [7] [8] [9] planteaban problemas de flexibilidad bastante evidentes desde el principio, ya que o bien su uso no era muy extenso y había poca información, o bien estaban encaminadas a juegos de arcade. Además, nos interesaba poder analizar bien nuestros datos, por lo que las herramientas de visualización que presentan estas librerías también era otro elemento de interés.

Eso nos dejaba con tres librerías entre las que elegir. Aquí se debe mencionar que los experimentos descritos en esta memoria, con casi total seguridad, se podrían realizar con cualquiera de ellas. Sin embargo, al final nos decantamos por ChainerRL debido a que permite implementar de forma sencilla y sin mucho esfuerzo agentes de variedad de tipos. En el artículo elaborado por J. Suarez-Varela *et al*. [1], sus autores hablan de esta librería y de sus capacidades y es la que emplean en su estudio, lo que también influyo en nuestra decisión, pues esta ha sido una de las referencias fundamentales de este trabajo.

Llegados a este punto podemos decidir si trabajaremos con aprendizaje profundo (*Deep Learning)* o no. En sistemas de RL básicos, se construyen tablas con información relativa a todos los estados del sistema. Como ya mencionamos en la sección 2.1, esto se hace con las funciones de valor. Si se conocen todos los estados del sistema (o es razonable alcanzarlos todos), se puede completar estas tablas, pero en general la mayoría de los problemas actuales que se intentan resolver son lo suficientemente complejos como para justificar el uso de técnicas de aprendizaje automático, y en particular el uso de redes neuronales profundas, para realizar una aproximación de la función de valor. La razón es que hay que generalizar estados porque hay demasiados (construir las tablas escala muy mal) y la información del entorno no es completamente conocida. Esto obliga a emplear la aproximación de funciones de valor (aprendizaje supervisado), lo que se resuelve, típicamente, con algoritmos de aprendizaje profundo. Para nuestro caso, tal como se verá más adelante, los sistemas serán potencialmente complejos por lo que se emplearán este tipo de algoritmos. En la librería ChainerRL, todos los algoritmos implementados ofrecidos usan aprendizaje profundo, por lo que podremos usarla.

Habiendo decidido qué librería usar para implementar agentes de RL, hay que ver cómo se hace esto en ChainerRL. La idea fundamental en esta librería es que se tienen distintas clases de Python que implementan a cada algoritmo. Todas incluyen un constructor que permite (con mucha flexibilidad de parámetros) instanciar un agente de RL implementado con el algoritmo en cuestión. Estas clases incluyen los métodos necesarios para entrenar y probar al agente.

Ya hemos hablado de los agentes y de cómo implementarlos, pero todavía falta por decir qué será de los entornos. Si bien ChainerRL permite implementar agentes y facilita su entrenamiento, no otorga funcionalidades para crear entornos de entrenamiento. Para ayudarnos con esto, la mayoría de las principales librerías de RL en la actualidad (entre ellas, ChainerRL) dan soporte a Gym [10], una librería abierta de OpenAI que implementa variedad de entornos de prueba. Lo cierto es que a nosotros no nos interesaba diseñar y probar algoritmos nuevos en entornos ya existentes, sino que queríamos usar un algoritmo ya existente para resolver un problema de establecimiento dinámico de conexiones. Esto significa que necesitábamos crear entornos personalizados para nuestras necesidades. Por suerte, Gym permite la creación de tales entornos, y gracias a esto es sencillo usarlos en conjunto con ChainerRL.

Para crear un entorno de Gym lo normal es seguir la estructura típica que estos presentan. Esto es, una clase con varios métodos obligatorios cada uno con su función respectiva. En concreto, se necesita un método de inicialización (constructor *init*) que crea e inicializa todas las variables de la clase. Estas variables serán las que luego se empleen para mantener el estado del entorno y calcular observaciones y recompensas al recibir una acción. Por otra parte, también se necesita un método de reinicio (*reset*). Este método se encarga de inicializar todas las variables del entorno para devolver a este al estado inicial. La importancia de esto es significativa ya que, aunque el método de inicialización puede poner

el entorno en el estado inicial, no se encarga de ello por definición. Además, es interesante poder reutilizar el mismo entorno ya creado para varios entrenamientos diferentes.

Pasemos ahora al método que avanza el estado del entorno con cada acción (*step*). Recibe una acción del agente, y a partir de ésta y de las variables que almacena calcula la observación y recompensa que se le devolverán al agente. Este proceso también puede modificar los valores de las variables del entorno. Este método es fundamentalmente el que se encarga de definir el funcionamiento del entorno y dependiendo de lo que se quiera que haga, se definirá de una manera u otra. Hay un método necesario a mayores en los entornos de Gym que se encarga de generar la información visual del estado del entorno. Es el método *render* y por definición debería devolver un vector que contenga una imagen RGB para poder visualizarla por pantalla. Como puede deducirse, esto no afecta al funcionamiento del entorno y al agente, es solamente información que puede observar el usuario. Como veremos más adelante, esto no nos es muy útil ya que la información que nos interesa no es visual, por lo que este método, aunque implementado, no se usa de la manera prevista por Gym. Asociado al correcto funcionamiento de la visualización del entorno esta también el método *close*. Nosotros no lo usaremos ya que al igual que antes, no es fundamental para el funcionamiento del entorno, siendo su principal función la de cerrar objetos que visualicen información del entorno con el método *render* (no se usa para eliminar el entorno en sí).

La instanciación del entorno requiere del uso de funciones definidas en la librería de Gym, si bien se debe crear un paquete Python local con una estructura concreta e instalarlo con la herramienta *pip* si se quiere usar entornos propios. Los detalles de cómo se hace esto se pueden ver en la sección correspondiente de [10]. Un último comentario es que, para ejecuciones sucesivas de entrenamientos, la creación del entorno de Gym personalizado no se puede hacer sin eliminarlo. Para automatizar esto, nosotros nos decantamos por eliminarlo en el *script* principal.

Volvamos ahora a donde lo habíamos dejado con ChainerRL. Sabiendo ahora cómo crear agentes y los entornos de entrenamiento que utilizarán debemos ver más en detalle qué agentes podemos usar. En [3] se muestra una tabla con todos los algoritmos disponibles, y también se puede acceder a la documentación de la librería. En esta documentación tenemos la API de la librería y podemos ver su código. Nosotros usaremos a lo largo de esta memoria cuatro algoritmos elegidos por diversas razones que pasaremos a detallar en las siguientes subsecciones.

2.2.2. Algoritmos de aprendizaje por refuerzo empleados

Antes de especificar algoritmos concretos deberíamos hablar de una técnica de aprendizaje por refuerzo denominada *Q-Learning*, en su variante de aprendizaje profundo, ya que será la base para tres de los cuatro algoritmos indicados a continuación. El *Q-learning* se basa en la función de valor de una acción (función Q). Esta función es similar a la función de valor de un estado, pero en lugar de indicar “lo bueno” que es estar en ese estado, indica “lo bueno” que es estar en un estado concreto y tomar una acción determinada (en realidad, indica el retorno esperado cuando se está en un estado concreto, se toma una acción determinada, y se sigue, a partir de ese momento, la política que se esté considerando). Como ya se mencionó antes, al estar hablando de aprendizaje profundo, esta función de valor de la acción se aproxima para generalizar distintos casos. En ChainerRL, esto se implementa por medio del algoritmo DQN (*Deep Q-Network*) [3]. Este algoritmo está basado en el artículo [11] y por medio de una red neuronal aproxima la función de valor de acción óptima.

La forma de instanciar un agente DQN en ChainerRL nos permite seleccionar la red neuronal que deseamos emplear, que definirá la función Q. Con las diferentes funcionalidades presentes en la librería se pueden definir redes neuronales como una clase o usar constructores para clases ya existentes que aceptan diversos parámetros para construir redes neuronales de distintos tipos. Otro argumento importante que requieren los agentes es un explorador. Con esto se define la política exploratoria del agente y para instanciar el objeto se pueden usar los constructores ya proporcionados por la librería. Hay una considerable cantidad de parámetros a mayores, aunque la mayoría tienen valores por defecto y se pueden omitir. Obviamente no ajustar estos parámetros puede llevar a resultados subóptimos en los experimentos, pero por brevedad nos centramos en cómo se define la función Q, la exploración y el factor de descuento, así como el intervalo de actualización (cuántas acciones realiza el agente antes de actualizar su modelo) y el tamaño inicial del buffer de *experience replay* [11] [12] (cuántas acciones debe realizar el agente antes de que se comience a emplear el mecanismo, suponiendo que el sistema se configura para almacenar una experiencia por cada acción tomada).

El mecanismo de *experience replay* tiene un impacto muy significativo en los resultados. Fundamentalmente se trata de un mecanismo que permite a los agentes acumular experiencias durante su entrenamiento para luego tenerlas en cuenta en el aprendizaje de los pesos de la red neuronal más adelante. Se define una experiencia como un conjunto de un estado inicial, la acción que lo sigue, el estado resultante y la recompensa obtenida. Estas experiencias acumuladas son dadas al algoritmo de aprendizaje de forma repetida, pero eligiéndolas aleatoriamente de la base de datos de experiencia, de modo que se evitan los problemas de correlación que existirían si se tomaran las experiencias de forma consecutiva.

También respecto a la actualización de los modelos en los agentes implementados por ChainerRL conviene mencionar que, estos admiten entrenamientos continuos o episódicos. La única diferencia es que, en caso de ser episódicos, se deberá usar un método especifico de entrenamiento que implementan los agentes al llegar al terminar un episodio.

Ahora mencionaremos brevemente los algoritmos utilizados para la realización de experimentos en esta memoria. La mayoría de estos son variantes del algoritmo DQN, por lo que sus argumentos de entrada son muy similares quitando alguna diferencia que describiremos.

2.2.2.1. DDQN (Double Deep Q-Network)

Comenzamos por la variante de DQN más parecida al mismo. El algoritmo DDQN [13] no cambia en nada los argumentos de entrada. Es una variante propuesta para resolver varios problemas detectados con el algoritmo DQN básico. Concretamente, este algoritmo reduce las frecuentes sobreestimaciones de valores de acción.

2.2.2.2. SARSA (State-Action-Reward-State-Action)

Al igual que antes, este algoritmo acepta los mismos argumentos que DQN. La diferencia fundamental entre DQN y SARSA [2] es que el primero aprende la función Q de la política óptima (*off-policy*), mientras que, canónicamente, el segundo aprende una política cercana a la óptima que viene definida por la exploración (*on-policy*). Esto implica que los parámetros de la exploración deberán ajustarse mejor y posiblemente sea importante que cambien durante el entrenamiento (por ejemplo, con probabilidad de exploración lineal decadente). Sin embargo, una ventaja bien conocida de SARSA es que tiene en cuenta malos resultados debidos a acciones exploratorias, mientras que DQN no. Esto permite evitar acciones arriesgadas y será de interés observar cómo afecta esto a los experimentos.

Dicho lo anterior, es importante mencionar que el algoritmo SARSA implementado en ChainerRL es *off-policy*. En concreto, es una variante de *Expected* SARSA [2], un algoritmo definido como *off-policy* debido a que en su caso la *target policy* y la *behaviour policy* no son iguales. El punto clave es que sigue teniendo en cuenta la exploración durante la evaluación de la política, simplemente la política evaluada no es la misma que la que se usa a la hora de tomar las acciones.

2.2.2.3. PAL (Persistent Advantage Learning)

Este es el último de los algoritmos que utilizaremos que comparte los argumentos de entrada del DQN. Sin embargo, para el algoritmo PAL [14] aparece un parámetro a mayores, el peso de las ventajas persistentes (α). Este algoritmo propone una modificación al operador de Bellman[1] que intenta mitigar inconsistencias entre la decisión sobre cuál es la acción óptima en un estado. Esto significa que incrementa la diferencia entre los distintos valores de la función Q en un estado para acciones distintas, diferenciando la óptima de las demás.

Como ya hemos mencionado antes, por simplicidad no nos centramos demasiado en la mayoría de los parámetros, así que el parámetro añadido del algoritmo PAL no se modificará respecto de su valor por defecto.

2.2.2.4. TRPO (Trust Region Policy Optimization)

El último algoritmo que usaremos en nuestros experimentos es el TRPO [15]. Es un método de gradiente de la política (*policy gradient*). Este tipo de algoritmos sigue una estrategia diferente de los anteriores, en lugar de aprender las funciones de valor (para luego tomar acciones basándose en las mismas), intenta aprender directamente una política parametrizada para que pueda seleccionar las acciones sin consultar una función de valor. Recordemos que una política es la probabilidad de elegir una acción determinada cuando se está en un determinado estado. Pues bien, estos algoritmos intentan proporcionar los valores de esas probabilidades directamente. Por estos motivos será el único cuya configuración varíe considerablemente del DQN. La principal la razón de esto es que en lugar de tomar una función Q como argumento de entrada que define la red neuronal, se le debe pasar una política a optimizar.

La implementación de TRPO en ChainerRL también requiere una función de valor de estado. Esto se debe a que se hace uso del mecanismo GAE ("*Generalized Advantage Estimation*") [16] para reducir la varianza de la estimación del gradiente, si bien aumenta un poco el sesgo.

Para este caso concreto no se modifican más parámetros y la configuración es básica por las mismas razones ya mencionadas anteriormente. Aun así, en nuestros experimentos definiremos la política de forma que esté implementada con una red neuronal con la misma estructura que en los otros algoritmos. Aparte, si bien es cierto que no modificaremos parámetros en general, el factor de descuento sí es común con los otros algoritmos y lo variaremos.

[1] Una forma de resolver las ecuaciones de Bellman para encontrar la política óptima se basa en el uso de una técnica llamada *Value Iteration*. Esta técnica consiste en aplicar iterativamente las ecuaciones de Bellman para ir actualizando sucesivamente (en cada iteración) la estimación de la función de valor. El operador de Bellman, es un operador matemático que realiza dicha actualización.

3. Búsqueda de la ruta de menor coste

En este capítulo nos centramos en el problema de enrutamiento en una red de comunicaciones mediante aprendizaje por refuerzo, para lo cual consideraremos la búsqueda de la ruta de menor coste entre dos nodos concretos de la red.

3.1. Alternativas para la búsqueda de la ruta de menor coste

Hablemos primero de otras alternativas ya existentes. Vamos a emplear dos soluciones ya implementadas para tomarlas como base y comparar con los resultados de nuestras propuestas. Serán el algoritmo de Dijkstra [17] y un algoritmo basado en *Q-learning*, llamado *Q-routing*. Este segundo mecanismo se caracteriza por buscar reducir el coste en lugar de maximizar la recompensa, y no usar un factor de descuento [18].

Con respecto al primer método, existe una librería de Python denominada Dijkstar [19] que implementa el algoritmo de Dijkstra y permite encontrar la mejor ruta en una red donde cada enlace entre dos nodos tiene un coste asociado no negativo. El resultado será pues, la ruta desde un nodo origen hasta otro destino definidos que sume el menor coste.

Con respecto al segundo método, es el presentado en [20]. El algoritmo está basado en RL, aunque no implementa ningún agente por medio de librerías como lo hacemos nosotros.

3.2. Agentes RL

El resultado final que buscamos es tener tres algoritmos diferentes que toman topologías de red con el mismo formato y encuentran la ruta menos costosa. Esta es una buena situación para establecer una comparación entre nuestra propuesta, usando las capacidades de ChainerRL y Gym, con respecto a la solución convencional del algoritmo de Dijkstra y a otra solución basada en RL. Así pues, debemos crear un agente con ChainerRL que, entrenando en el entorno de Gym que proponemos a continuación, encuentre la ruta menos costosa.

La creación de un agente con ChainerRL resulta fácil, y como es compatible con entornos de Gym, no hay mayores dificultades. Únicamente hay que emplear las herramientas proporcionadas por la librería como ya se explicó en la sección 2.2.1.

En este capítulo solamente usaremos uno de los cuatro algoritmos introducidos en la sección 2.2.2, siendo este el DDQN. Respecto a sus hiperparámetros, para el factor de descuento γ y el tipo de explorador, así como su probabilidad de exploración, realizaremos barridos de parámetros para ver los resultados con distintos valores. Esto nos permitirá decidir unos valores aproximadamente óptimos.

Los demás hiperparámetros serán los de por defecto que vienen definidos en ChainerRL, salvo por dos excepciones. Se trata del intervalo de actualización de los parámetros de la red neuronal que proporcionan la estimación del valor objetivo (*target*) durante el aprendizaje y del tamaño inicial del buffer de *experience replay*. Por defecto, el valor del intervalo de actualización es de 10000 (tras 10000 pasos el agente actualiza la red neuronal que usa para dar las estimaciones de la función Q *target*), mientras que el tamaño del buffer de *experience replay* es de 50000 (solo comienza a usar el mecanismo una vez ha almacenado 50000 experiencias). Como describiremos a continuación, nosotros no realizaremos entrenamientos lo suficientemente largos en este capítulo como para usar valores tan altos, por lo que los reducimos a 100 y 500, respectivamente para el intervalo de actualización y el buffer inicial de *experience replay*.

3.3. Entorno

3.3.1. Ideas generales

Vamos a crear un entorno de Gym que cargue parámetros de forma que podamos fácilmente probar nuestros algoritmos con distintas topologías de red. Esto es útil a la hora de comparar la solución que planteemos con las alternativas convencionales.

Vamos a definir la topología de red por medio de enlaces que conecten los nodos de la red, y tengan asociado un coste, así que el entorno de Gym pertinente se crea con esto en mente.

Crearemos un entorno muy similar al del problema clásico de resolver un laberinto con RL. Habrá N nodos en la red (numeramos los nodos con $n = 0, \dots, N-1$). Un agente se irá moviendo por los nodos de la red, desde el nodo origen de la conexión s, hasta el nodo destino d. El objetivo es que el agente alcance el nodo destino de la conexión que se desea establecer (lo cual sería análogo a llegar a la salida del laberinto). El estado en el que se encuentra el agente coincide con el nodo en el que está en ese momento. Como tendremos un nodo destino, el estado correspondiente a este será el terminal, por lo que estaremos ante una tarea episódica.

Para moverse entre los nodos (estados) tendremos los enlaces que los conectan (los numeramos con e). Por simplicidad asumiremos que los enlaces son unidireccionales. El agente tomará una acción en cada estado (esto es, en cada nodo) que implicará elegir uno de los enlaces por los que salir. Todos los enlaces tendrán costes asociados (c_e, con c_e ³ 0) que, al usarlos, se sumarán a un total. Minimizar este total conllevará minimizar el coste. No imponemos restricciones sobre el destino de los enlaces (pueden llevar a cualquier otro nodo incluyendo el mismo del que salen, dependiendo de la topología planteada) ni sobre cuántos hay en la red.

El resultado final es análogo a un laberinto de nodos que están interconectados por enlaces de forma desconocida al agente. Querremos pues que este encuentre el camino de menor coste al nodo destino de la conexión. Pero ¿cuáles serán las observaciones y recompensas devueltas por el entorno?

3.3.2. Observación

En el caso de la observación, coincidirá con el estado, así que usamos un vector binario con tantos elementos como nodos en la red analizada haya. Esto es así ya que hemos definido que estar en cada nodo significa estar en un estado distinto, por lo que cada elemento del vector indicará un estado. Podemos indicar esto al agente de forma clara haciendo que los valores de estos elementos sean 0 si en el estado actual no nos encontramos en dicho nodo, y 1 si así es, es decir, se ha usado una codificación *one-hot* [21]. La razón de emplear esta codificación es que el nombre o el número de nodo es realmente una variable categórica y no hay una relación de orden entre los nodos. Una vez definida la codificación, el agente puede ver fácilmente el estado de la red, y relaciona rápidamente qué acciones son las más acertadas en cada nodo de la red. Esto es importante debido a que las acciones en cada nodo no cambian. Estando en un nodo concreto de la red, tomar el enlace asociado a la acción "0" puede ser una buena idea, pero ya no tiene por qué ser así en los demás. Hay que recordar que el agente tendrá un número de acciones fijo y que no cambiarán en función del estado en el que se encuentre (pero sus resultados sí). Es por eso por lo que la observación debe reflejar de manera muy concreta en qué nodo se encuentra el agente que está buscando la ruta menos costosa.

Otras observaciones razonables como, por ejemplo, incluir los costes de los enlaces o la topología de la red no son buenas porque lo único necesario en este caso es mostrar el

estado en el que se encuentra el entorno. Este tipo de observaciones también pueden cumplir este cometido, pero lo hacen de manera más compleja innecesariamente.

3.3.3. Acciones

Acabamos de hablar de las observaciones y hemos mencionado las acciones que puede tomar el agente, así que profundicemos más en esto último. Debido a la forma en la que se definen las funciones Q y las políticas en ChainerRL, los agentes tienen un número invariante de acciones posibles que pueden tomar. Lo normal es que cada acción implique la elección de un enlace saliente desde un nodo concreto. En nuestro entorno, el agente comienza un episodio de entrenamiento desde un nodo inicial de la red (s) y en cada paso toma una acción que le lleva a otro. Este proceso continúa hasta que se alcanza el nodo destino (d), momento en el que termina el episodio de entrenamiento. Plantear los episodios de esta manera es útil porque es razonable pensar que cuanto mejor aprenda el agente a encontrar el nodo destino, más cortos serán los episodios de entrenamiento y más rápido será el aprendizaje.

El problema al que nos enfrentamos es que nuestros agentes tienen siempre el mismo número de acciones posibles en cada nodo de la red. Esto ocurre porque estamos planteando una solución de aprendizaje profundo (y por tanto basada en aproximación de funciones de valor) en lugar de tabular. En el caso de que la topología de red tenga un número variable de enlaces salientes desde cada nodo, es decir, no todos los nodos tengan el mismo número de enlaces de salida, se presentaría el problema mencionado (pues el número real de acciones posibles en cada nodo sería distinto). En un caso tabular no habría problema porque la tabla que se define puede tener esto en cuenta. Sin embargo y para nuestro caso, el agente tiene un número de acciones fijo para todos los estados.

Si hay menos acciones posibles que el número de enlaces salientes máximo en cualquier nodo de la red, el agente será incapaz de usar alguno de estos enlaces. Esto no se puede ignorar, así que la única solución razonable es establecer como número de acciones este máximo de la red. Ahora podríamos tener el caso contrario, que el agente tenga más acciones posibles que enlaces disponibles en un nodo. La solución que planteamos no es perfecta pero funcional. Consiste en que las acciones se mapean a los enlaces en los nodos de forma creciente. En el caso de que un nodo tenga menos enlaces salientes que acciones disponibles, las últimas acciones se considerarán como sobrantes. Si el agente toma una de estas acciones, se considerará como un paso en falso y el estado de la red se mantendrá igual (es decir, se permanece en el mismo nodo). A mayores, habrá una penalización por tomar estas acciones para que el agente aprenda a no usarlas. Con esto básicamente estamos añadiendo enlaces salientes a los nodos que no alcancen el máximo de la red, cuyos destinos son el mismo nodo del que salen y tienen un coste concreto asociado.

Como ya hemos dicho, este planteamiento no es ideal. Estamos dejando en manos del agente el aprender que acciones son válidas en cada estado. Se pueden encontrar discusiones sobre esta problemática, por ejemplo, en [22]. El principal problema que se observa al utilizar nuestra solución en estos casos es que el aprendizaje es más lento, una posible mejora es simplemente mapear las acciones sobrantes a otras siempre presentes para que de igual cuál de estas elija el agente. Sin embargo, esto sigue siendo un planteamiento muy similar y mejora poco sobre lo descrito. Otras soluciones plantean especializar a las estructuras de las redes neuronales de forma que automáticamente su probabilidad de elegir una acción inválida sea 0. Esto es obviamente una solución no disponible al utilizar una librería ya definida. Por último, otra posibilidad podría ser transformar el espacio de acción en función del estado para que siempre sea fijo a la entrada del entorno proveniente desde el agente.

El problema de esto es que es extremadamente difícil dar una transformación que sea utilizable.

Entonces, ¿cómo sabemos que nuestra solución es aceptable en este caso? Porque nuestro espacio de acción es relativamente pequeño. Es cierto que todo esto es un problema considerable que afecta a al aprendizaje profundo RL debido al uso de redes neuronales cuya estructura se define fija normalmente. Sin embargo, nosotros hablamos de unos pocos enlaces salientes (es decir, unas pocas acciones), lo que no es un espacio de acción muy grande. Es razonable que al resolver un problema sencillo también se busque una solución sencilla. Es por esto por lo que nuestro planteamiento será suficiente, aunque siguiendo esa misma línea, usar un enfoque tabular sería aún más sencillo y eficiente.

3.3.4. Recompensa

Pasemos ahora al último elemento fundamental del entorno, las recompensas. Nos preguntamos qué recompensas dar al agente por las acciones que va tomando mientras busca el nodo destino. Obviamente estas recompensas deberán estar ligadas a los costes de los enlaces, pero la pregunta es cómo pasarlas al agente.

Nuestro planteamiento consiste en otorgar una recompensa negativa (penalización) inmediata por cada acción tomada, proporcional al coste del enlace que se hay usado (en nuestro caso es el coste directamente). Si bien ahora la recompensa acumulada final será negativa, esto sigue siendo válido y el agente buscará la mayor recompensa de todos modos, que será análogo a buscar el menor coste.

3.4. Experimentos

En este apartado veremos varios experimentos distintos que nos permitirán obtener una solución al problema planteado. El objetivo no es realmente obtener una solución mejor a la que se obtiene usando los algoritmos alternativos con los que la compararemos, sino familiarizarnos con el uso de ChainerRL y Gym, así como de RL en general, aplicándolo a un caso práctico.

En estos experimentos presentaremos datos en forma de tablas y gráficas, para los que es importante que definamos varios conceptos. Llamaremos tasa de acierto al porcentaje de las simulaciones en las que el algoritmo empleado obtiene la solución óptima. Para calcular este valor se entrenarán agentes idénticos en condiciones iniciales iguales, y de estos se tomará el resultado. A mayores, para saber si un agente ha encontrado la solución óptima o no, se le pide que después del entrenamiento vuelva a realizar un episodio de entrenamiento sin exploración, dando como resultado la ruta que dicho agente considera como la mejor posible (y que compararemos con la óptima). Otra métrica de interés será el tiempo medio de entrenamiento, que se calculará como el promedio de los tiempos de entrenamiento de varios agentes iguales en condiciones iniciales idénticas.

En lo que se refiere a las gráficas que se observarán en este apartado, representarán la evolución de las recompensas inmediatas acumuladas en los episodios de entrenamiento que van obteniendo los agentes, promediadas con una ventana deslizante de cien valores. Con esto nos referimos a que se tomarán las últimas cien recompensas inmediatas acumuladas en los episodios de entrenamiento obtenidas por un agente (si aún no ha obtenido cien, se toman todas) y se promedia su valor para obtener un punto en la gráfica. Puesto que se trata de recompensas inmediatas acumuladas, es importante recalcar que en el cálculo de dicho valor no se tiene en cuenta el valor del factor de descuento, γ.

3.4.1. Experimento en red con bifurcación

En nuestro primer experimento planteamos una topología de red que permita diferenciar claramente entre la ruta más corta y la de menor coste. Para explicar esto fijémonos en la siguiente red (Figura 1). El objetivo del agente será, comenzando en el nodo $s = 1$, llegar al $d = 8$.

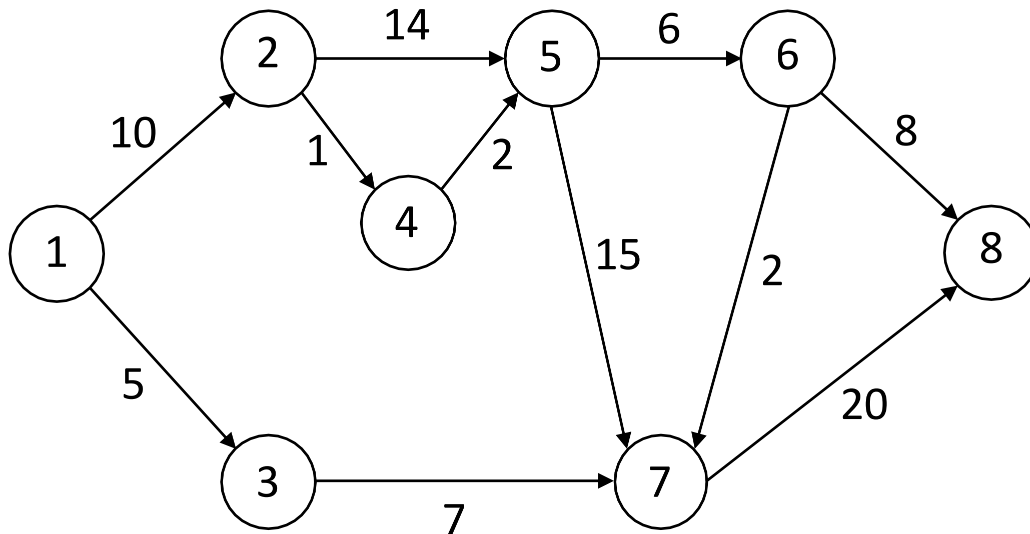

Figura 1 – Red con bifurcación

Como se ve, inmediatamente al comenzar la búsqueda el agente debe tomar una decisión importante, tomar la ruta de abajo o la de arriba. El camino óptimo se encuentra por arriba (ruta 1-2-4-5-6-8), pero por abajo se encuentra la ruta más corta. Debido a la naturaleza de la exploración aleatoria, el agente encontrará con más frecuencia el nodo destino eligiendo el camino de abajo. Esto implica que, en episodios posteriores del entrenamiento, si no toma una acción aleatoria en el primer nodo de la red, se decantará por el camino subóptimo, dificultándole encontrar la verdadera mejor ruta. Además, la red está diseñada de modo que, aunque la ruta óptima se encuentra por la parte superior de la topología, si el agente se mueve aleatoriamente por esa zona superior de la red, es muy probable que acabe yendo por una ruta de mayor coste que si va por la parte de abajo, lo cual dificulta que encuentre la ruta óptima. Se trata por tanto de una topología que pone las cosas difíciles al mecanismo de RL, y dicha dificultad se ve aumentada además por el hecho de utilizar la estrategia de aproximación de funciones en lugar de emplear una estrategia tabular como ya se ha mencionado anteriormente.

Por último, es importante recordar que los agentes pueden dar "pasos en falso" (pues se ha supuesto que en todos los estados hay dos acciones posibles, pero hay nodos con un solo enlace saliente) y estos "pasos en falso" serán penalizados de manera considerable. Es importante que estas penalizaciones sean comparables con los costes de los enlaces para que los agentes los eviten. Fijándonos en los costes de los enlaces de la Figura 1 decidimos que un coste de 10 para estas acciones podría ser adecuado (ya que es un valor aproximadamente intermedio), de modo que es el empleado en los experimentos.

Veamos entonces algunos resultados de nuestro algoritmo con la red de la Figura 1. Para ello usaremos varios agentes con distintos parámetros y compararemos sus resultados. Como se trata de un experimento básico no usaremos más que el algoritmo DDQN explicado en la sección 2.2.2.1. Para ello implementaremos una red neuronal con tantos nodos de

entrada como elementos tenga el vector de observación, una capa oculta de 30 nodos y tantos nodos de salida como acciones máximas hay, esto es, dos. Las conexiones entre capas son lineales con todos los nodos teniendo enlaces con todos de las capas adyacentes. Como función de activación usamos la tangente hiperbólica. Empecemos por buscar el mejor valor para el factor de descuento.

En la prueba inicial se han instanciado 11 tipos de agentes distintos. Cada tipo trabaja con un factor de descuento distinto, que varía en pasos de 0.1 y va desde 0 hasta 1. Además, cada tipo de agente estará replicado cinco veces para afianzar los resultados (es decir, se crean cinco agentes iguales pero independientes que se entrenan por separado, como si de cinco simulaciones diferentes se tratara). Todos incluyen exploración aleatoria con probabilidad de exploración 0.2 en cada decisión que toman. Buscamos el valor del factor de descuento (γ) con el que se aprende la ruta óptima entre los nodos 1 y 8 de la red (la secuencia 1, 2, 4, 5, 6, 8) en el menor tiempo posible. Fijamos el final del entrenamiento como el instante en el que se completan 1000 episodios por lo que los tiempos no son representativos de lo que tardan realmente los agentes en alcanzar la solución. Sin embargo, cuanto antes lo hagan menos tardaran en completar el entrenamiento, por lo que estos tiempos nos sirven para comparar estos agentes entre sí. A continuación, mostramos los resultados obtenidos:

Gamma (γ)	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1**
Tasa de acierto (%)	0	0	0	0	0	100	100	100	100	100	40
Tiempo medio (s)	33.7	33.6	35.1	34.5	33.4	17.2	16.4	16.0	16.4	15.1	17.1

Tabla 2 – Comparación amplia de valores del factor de descuento para red con bifurcación

Como se puede ver en la Tabla 2, el mejor valor para el factor de descuento estará entre los valores de 0.5 y 0.9, que no solo presentan la mejor tasa de acierto, sino que también tardan menos que los demás en completar el entrenamiento. Una tasa de acierto del 100% para $\gamma = 0.5$ quiere decir que los 5 agentes que se entrenaron con ese valor del factor de descuento consiguieron aprender la ruta óptima tras los 1000 episodios. Aunque es cierto que al haber solamente cinco simulaciones para cada valor de γ distinto, la granularidad de estos resultados es muy baja, nos sirve al menos para hacernos una idea de cómo de buenos son los valores. Recordemos que el objetivo de estos experimentos es fundamentalmente la familiarización con el entorno de trabajo.

En la Figura 2 podemos ver una gráfica que muestra la evolución de los valores promediados con una ventana deslizante de tamaño 100 episodios de los valores de las recompensas inmediatas acumuladas (en dichos episodios) que obtienen las mejores réplicas de los distintos tipos de agentes. Decidir cuál de las réplicas ha sido la mejor se hace comprobando cuál de ellas ha obtenido a lo largo del entrenamiento mayores recompensas (se integran los valores almacenados para la gráfica, y nos quedamos solamente con los que proporcionan un mayor valor). En la figura también se muestra la recompensa óptima (correspondiente a la que se obtendría si se siguiera la ruta de menor coste). Es importante notar que, dado que se realiza exploración, no siempre se elige la acción que el algoritmo

considera óptima en cada paso, de modo que al promediar los resultados de 100 episodios la curva no alcanza ese valor óptimo (pues incluye rutas “exploratorias” no óptimas).

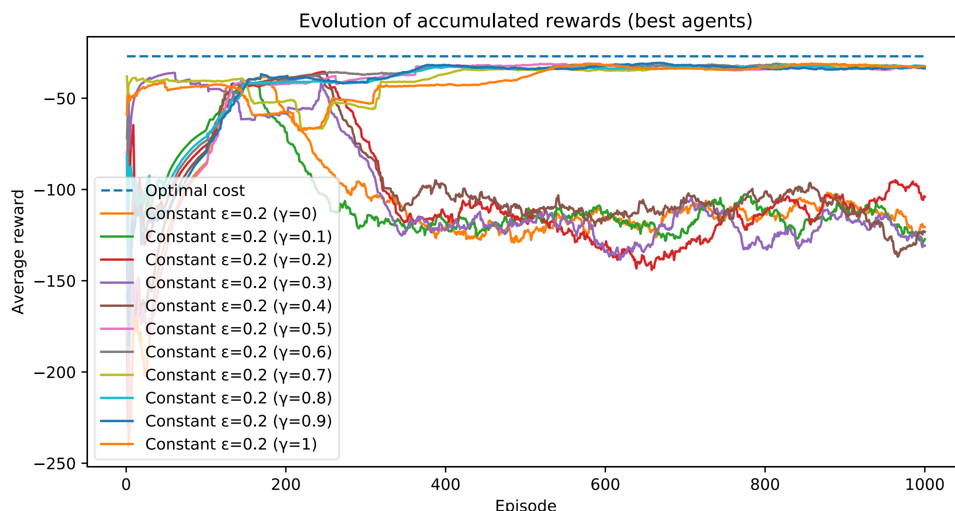

Figura 2 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios, para los mejores agentes con factores de descuento distintos en la comparación amplia para la red con bifurcación

Antes de decidirnos por un valor, probemos a analizar más a fondo lo mismo. Ahora mostramos los mismos resultados con valores que van desde 0.6 hasta 0.8 en pasos de 0.025. Elegimos descartar los valores cercanos a 0.5 y 0.9 ya que están cerca de los límites donde se comienzan a observar malos resultados. Como en estos experimentos no replicamos los agentes lo suficiente como para asegurar que la varianza sea muy baja, es perfectamente posible que al repetir el experimento se puedan observar tasas de acierto menores al 100% para valores de γ de 0.5 y 0.9 (aunque en los resultados mostrados en la tabla esto no ha sucedido).

Gamma (γ)	**0.6**	**0.625**	**0.65**	**0.675**	**0.7**	**0.725**	**0.75**	**0.775**	**0.8**
Tasa de acierto (%)	100	100	100	100	100	100	100	100	100
Tiempo medio (s)	13.7	14.3	15.2	16.1	15.9	15.3	15.7	17.0	16.1

Tabla 3 – Comparación detallada de valores del factor de descuento para red con bifurcación

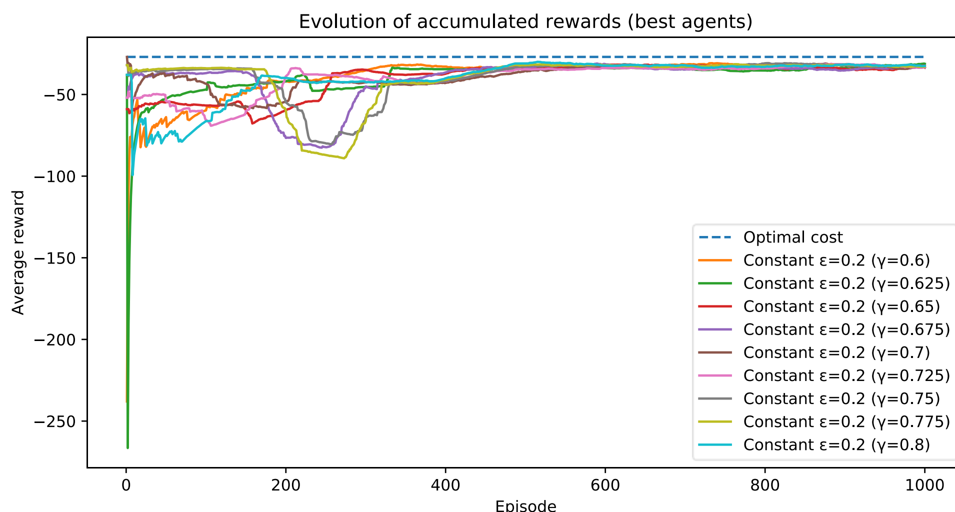

Figura 3 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con factores de descuento distintos en la comparación detallada para la red con bifurcación

Observando los resultados de la Tabla 3 decidimos usar un valor de factor de descuento de 0.7 para el resto de las pruebas en esta topología de red. Nos decantamos por este valor porque está en medio del intervalo de valores aceptables que hemos observado y no se ven diferencias destacables entre estos. Ahora con la misma metodología buscaremos encontrar la mejor exploración posible. Para ello primero vamos a buscar el mejor resultado con exploración aleatoria con probabilidad constante. Los resultados de dicho experimento para valores desde 0.05 hasta 0.4 en saltos de 0.05 se encuentran a continuación.

Epsilon (ε)	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4
Tasa de acierto (%)	80	100	100	100	100	100	100	100
Tiempo medio (s)	16.4	13.6	17.0	15.6	16.5	15.2	16.2	15.7

Tabla 4 – Comparación de valores de probabilidad de exploración constante con γ = 0.7 para red con bifurcación

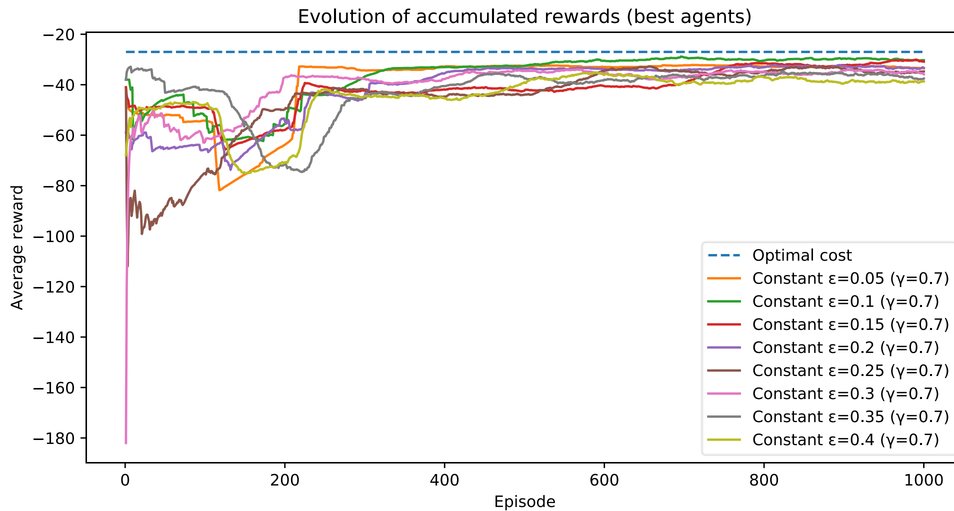

Figura 4 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con probabilidades de exploración constantes distintas con γ = 0.7 para red con bifurcación

De los resultados de la Tabla 4 y la Figura 4 deducimos que, para esta topología de red distintos valores de probabilidad de exploración constante proporcionan resultados muy similares (siempre que estos valores sean razonables, es decir, que no sean cercanos a 1). Si que es cierto que también se debe evitar una probabilidad de exploración demasiado baja, tal como se ve para el valor de 0.05. Es razonable pensar que, si esta probabilidad es demasiado baja, 1000 episodios de entrenamiento no serán suficientes para encontrar la ruta más corta.

Para la comparación posterior con los algoritmos convencionales escogeremos el mejor valor (0.1 por haber obtenido el menor tiempo de entrenamiento), pero cualquiera de ellos sería válido, sobre todo si tenemos en cuenta que a mayor probabilidad de exploración menor recompensa promedio debido a las acciones aleatorias más frecuentes.

Aún nos falta un tipo de exploración interesante, el lineal decadente. Es similar al anterior pero la probabilidad de exploración en cada momento no se mantiene constante, sino que decrece desde un valor inicial hasta uno final en un cierto número de pasos. Al igual que antes compararemos distintos valores posibles para buscar el mejor. Los valores comparados variarán sus probabilidades desde valores de 0.4 o 0.2 hasta 0.05, y lo harán en 5000, 10000, 15000 y 20000 pasos.

Epsilons (ε) inicial y pasos de decaimien-to	**0.4 \ 5000**	**0.4 \ 10000**	**0.4 \ 15000**	**0.4 \ 20000**	**0.2 \ 5000**	**0.2 \ 10000**	**0.2 \ 15000**	**0.2 \ 20000**
Tasa de acierto (%)	100	100	100	100	100	100	100	100
Tiempo medio (seg)	14.6	15.3	15.9	15.8	16.4	16.2	15.7	15.6

Tabla 5 – Comparación de valores de probabilidad de exploración decadente con γ = 0.7 para la red con bifurcación

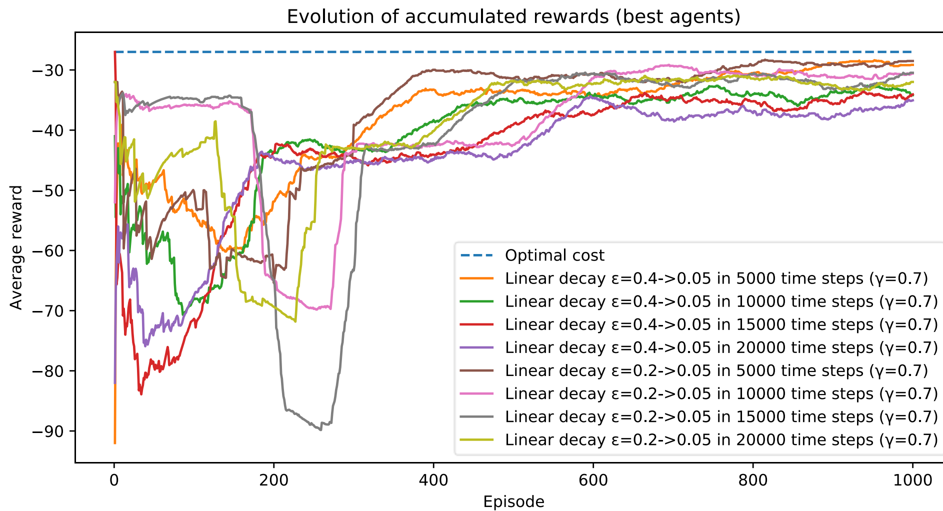

Figura 5 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con probabilidades de exploración lineales decadentes distintas con γ = 0.7 para la red con bifurcación

Observamos de los datos en la Tabla 5 y la Figura 5 que al igual que en el caso de la exploración con probabilidad constante, aquí los resultados tampoco cambian mucho. El mejor resultado lo obtenemos con una probabilidad inicial de 0.4 que decae a 0.05 en 5000 pasos, pero repitiendo el experimento esto cambia fácilmente.

Después de analizar diversos algoritmos que tenían distintos parámetros vamos a comparar los mejores con las dos métricas de las que ya hemos hablado antes. En la Tabla 6 vemos las diferencias.

Algoritmo	**Dijkstra**	***Q-routing***	**Basado en ChainerRL (exploración constante)**	**Basado en ChainerRL (exploración lineal decadente)**
Tasa de acierto (%)	100	0	100	100
Tiempo medio (s)	0.0	-	13.6	14.6

Tabla 6 – Comparación entre distintos algoritmos para la red con bifurcación

Obviamente como la red analizada en la Figura 1 tiene muy pocos nodos, el algoritmo de Dijkstra encuentra la solución óptima y lo hace, además, en un tiempo tan pequeño que no pudo ser medido y aproximaba a 0 (la diferencia entre la hora en milisegundos a la que comienza a calcular la ruta y la hora a la que acaba era demasiado pequeña para la precisión del valor devuelto por la función pertinente en Python).

Para el algoritmo *Q-routing* presentado en [20] tenemos un caso interesante. Después de varias pruebas, una de las cuales duro 7 horas y media, no alcanzo ninguna solución. Como nuestro objetivo de estudio no es este algoritmo en concreto, no vamos a comprobar exactamente la razón por la que falla, pero claramente no es un algoritmo robusto.

Por otra parte, nuestros agentes de ChainerRL son robustos (y si su configuración es adecuada encuentran la ruta óptima), pero los tiempos de entrenamiento no son comparables con el tiempo que tarda en encontrar el algoritmo de Dijkstra la solución. Esto es de esperar ya que en un caso estamos midiendo el tiempo de entrenamiento para con bastante certeza alcanzar la solución óptima, y en el otro sencillamente es el tiempo que se tarda en calcular la ruta de menor coste. Aunque es cierto que con ajustes sobre la forma en la que se entrenan podríamos sacar una métrica adecuada del tiempo que tardan en encontrar una solución, esto no interesa demasiado ya que es de esperar que en este caso el algoritmo de Dijkstra sea imbatible.

3.4.2. Experimento en red con sumidero

Como las pruebas realizadas sobre la anterior red no han sido del todo conclusivas, y además interesa tener más datos en nuestro estudio, probaremos ahora con otra red de ejemplo.

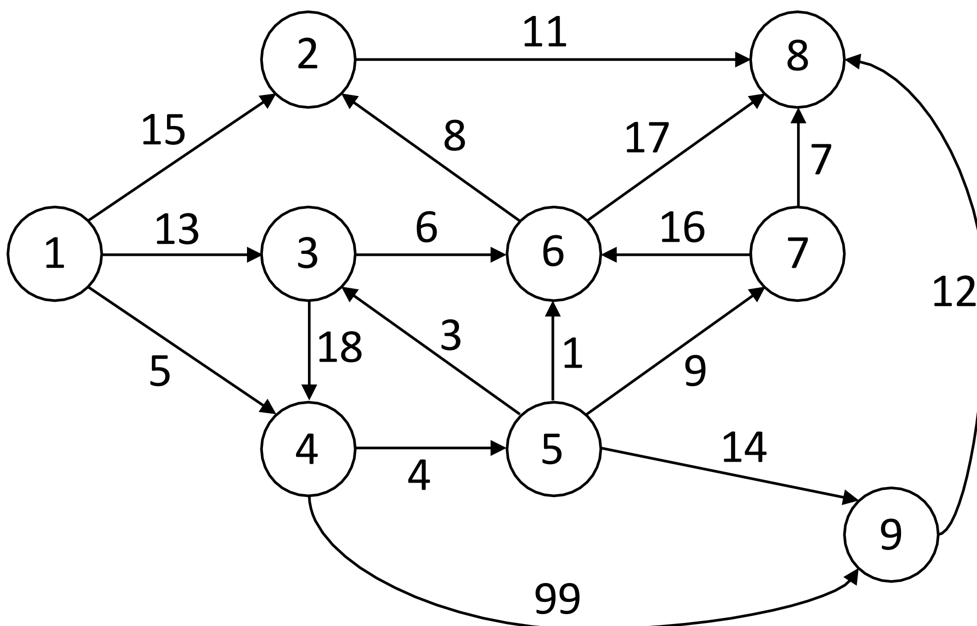

Figura 6 – Red con sumidero

Si en la red de la Figura 6 tomamos como nodo inicial el 1 y como nodo final el 9, en el nodo 8 tenemos un sumidero. Sería interesante ver cómo lidia nuestro algoritmo con esto. Además, esperamos que ahora el algoritmo *Q-routing* [20] sí funcione y sirva de comparación. Veamos los resultados de la topología de la Figura 6.

Para esta red volveremos a realizar el mismo proceso que hemos descrito antes. Esto significa que nuevamente hallaremos el mejor valor del factor de descuento (asumiendo exploración constante con probabilidad 0.2), así como los mejores agentes con exploraciones constante y lineal decadente, una vez determinado el factor de descuento a utilizar. Comenzaremos suponiendo exploración constante.

Gamma (γ)	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Tasa de acierto (%)	0	0	0	0	0	0	100	100	80	60	0
Tiempo medio (s)	143.5	143.6	144.5	145.8	143.7	140.7	70.8	53.6	84.1	89.6	127.0

Tabla 7 – Comparación amplia de valores del factor de descuento para red con sumidero

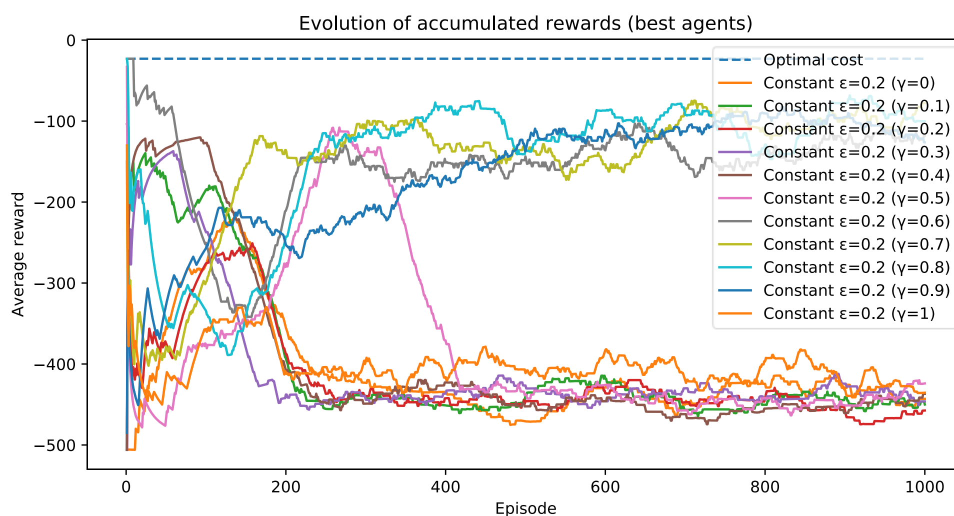

Figura 7 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con factores de descuento distintos en la comparación amplia para la red con sumidero

Gamma (γ)	0.6	0.625	0.65	0.675	0.7	0.725	0.75	0.775	0.8
Tasa de acierto (%)	100	100	100	100	100	100	100	100	100
Tiempo medio (s)	68.1	69.2	50.3	62.5	63.4	59.5	62.1	46.9	54.8

Tabla 8 – Comparación detallada de valores del factor de descuento para red con sumidero

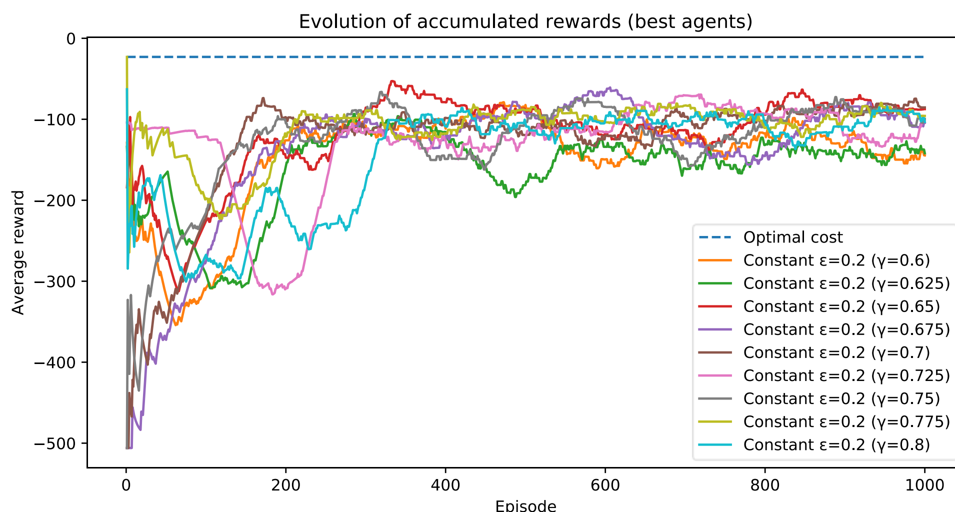

Figura 8 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con factores de descuento distintos en la comparación detallada para la red con sumidero

En las Tabla 7 y Tabla 8, y las Figura 7 y Figura 8 vemos resultados similares a los que observábamos en la topología de red de la Figura 1, solo que en este caso el intervalo de valores del factor de descuento aceptables parece ser más estricto. Sin embargo, vemos que, de nuevo, el valor de $\gamma = 0.7$ obtiene unos resultados buenos de manera fiable, lo cual es positivo porque nos gustaría poder usar una configuración idéntica del algoritmo sin importar la topología de la red.

Por otro lado, vemos que en esta red los tiempos de entrenamiento y tasas de acierto son peores que en la topología anterior. De esto deducimos que esta red es más difícil de resolver para nuestro algoritmo. La razón de esto es debido a que ahora hay más posibles acciones por nodo que los agentes pueden tomar (cuatro mientras antes eran solamente dos). Aun así, claramente es posible encontrar la solución adecuada. Veamos los resultados de los distintos tipos de exploración.

Epsilon (ε)	**0.05**	**0.1**	**0.15**	**0.2**	**0.25**	**0.3**	**0.35**	**0.4**
Tasa de acierto (%)	80	100	100	100	100	100	100	100
Tiempo medio (s)	103.2	53.3	50.2	51.5	67.0	75.5	61.4	69.4

Tabla 9 – Comparación de valores de probabilidad de exploración constante con γ = 0.7 para la red con sumidero

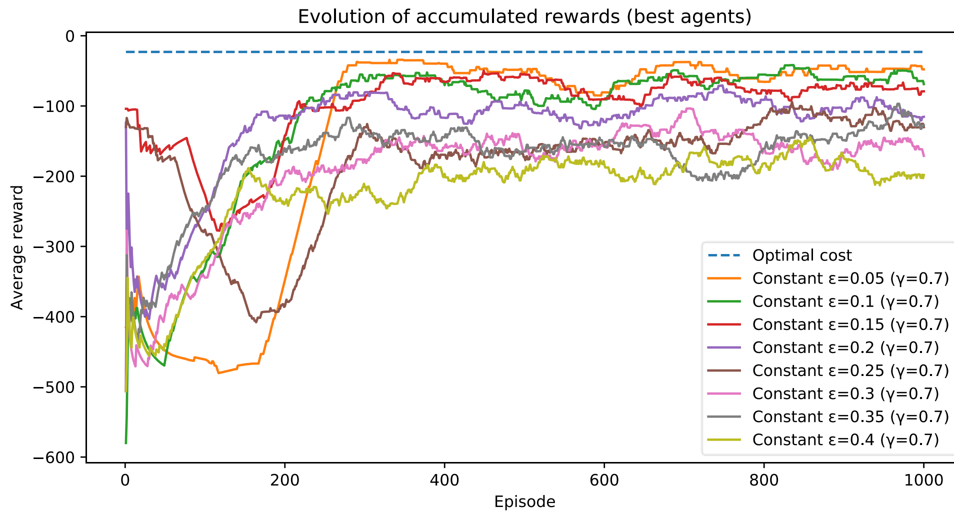

Figura 9 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con probabilidades de exploración constantes distintas con γ = 0.7 para la red con sumidero

De nuevo vemos que según los datos de la Tabla 9, la probabilidad de exploración constante no aporta resultados muy variados. Hay que mencionar que, aunque en la Figura 9 puede parecer que los agentes con probabilidad de exploración mayor obtienen peores resultados en cuanto a recompensas se refiere, esto se debe a que el promediado está teniendo en cuenta casos en los que las recompensas son muy malas debido a una mala acción aleatoria. Cuanto mayor sea la probabilidad de exploración, menores son los valores a los que tienden las curvas. Dicho esto, para los valores de probabilidad de exploración de 0.3, 0.35 y 0.4 parece que las curvas tienden a la baja más de la cuenta, indicando que valores por encima de 0.3 no son muy adecuados.

Como los mejores resultados los obtenemos con las probabilidades de 0.1, 0.15 y 0.2, volveremos a tomar 0.1 como el mejor valor para ser coherentes con el experimento de la red de la Figura 1.

Veamos por último el caso de la exploración lineal decadente en la Tabla 10 y la Figura 10.

Epsilons (ε) inicial y pasos de decaimiento	**0.4 \ 5000**	**0.4 \ 10000**	**0.4 \ 15000**	**0.4 \ 20000**	**0.2 \ 5000**	**0.2 \ 10000**	**0.2 \ 15000**	**0.2 \ 20000**
Tasa de acierto (%)	100	100	100	100	80	100	100	100
Tiempo medio (seg)	55.1	42.9	69.1	58.7	87.6	58.5	60.8	72.7

Tabla 10 – Comparación de valores de probabilidad de exploración decadentes con γ = 0.7 para la red con sumidero

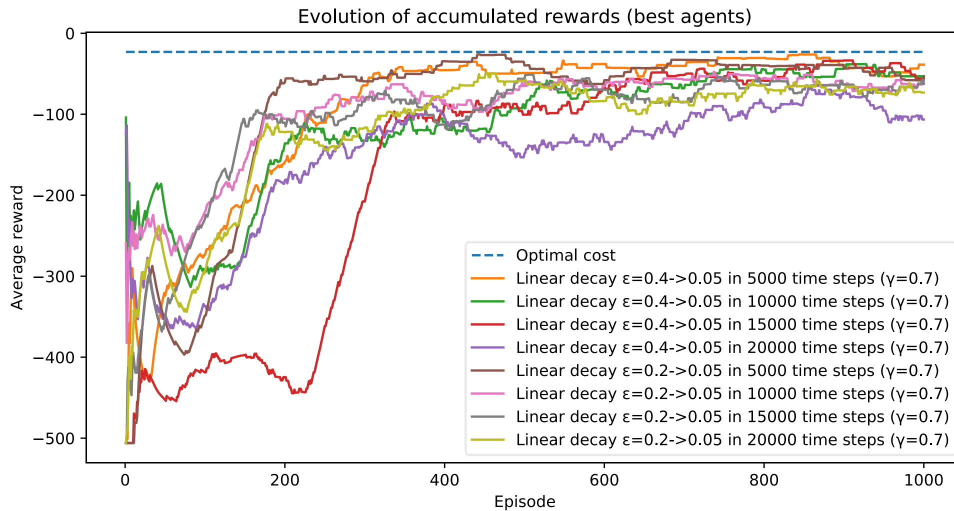

Figura 10 – Recompensas inmediatas acumuladas y promediadas con una ventana de 100 episodios de los mejores agentes con probabilidades de exploración lineales decadentes distintas con γ = 0.7 para la red con sumidero

Vemos que la exploración lineal decadente no mejora mucho respecto a la constante dando como resultado tasas de acierto y tiempos de entrenamiento similares. Lo único destacable es que los tiempos de entrenamiento de la Tabla 10 son algo variantes, lo que es un indicador de que la varianza con este tipo de exploración es algo alta, y concretamente mayor a la presente con exploración constante (Tabla 9). Esto indica que no merece la pena ajustar los parámetros de la exploración lineal para obtener buenos resultados teniendo a nuestra disposición la exploración constante que, de manera más fácil, nos da los aparentemente mejores resultados para este caso de uso concreto. Igual que antes volvamos a tomar como valores finales los correspondientes a la exploración decadente lineal que comienza en 0.4 y finaliza en 0.05 en 5000 pasos, ya que es el valor que usamos en el experimento de la Figura 1.

Para concluir el análisis de la topología de red de la Figura 6, vamos a comparar todos los algoritmos igual que hicimos anteriormente.

Algoritmo	**Dijkstra**	***Q-routing***	**Basado en ChainerRL (exploración constante)**	**Basado en ChainerRL (exploración lineal decadente)**
Tasa de acierto (%)	100	100	100	100
Tiempo medio (s)	0.0	0.016	53.3	55.1

Tabla 11 – Comparación entre distintos algoritmos para la red de la figura 8

Respecto a nuestra comparación en la red de la Figura 1, en este caso el algoritmo de Dijkstra es el claro ganador otra vez. Aunque la red de la Figura 6 sea algo más compleja, sigue teniendo solamente 9 nodos, lo que es fácil de resolver matemáticamente. En cuanto al algoritmo de *Q-routing*, esta vez sí encuentra una solución y lo hace además en un tiempo

muy bajo. Este algoritmo rellena una tabla (enfoque tabular) con las distintas acciones en cada estado, mientras que los agentes de ChainerRL utilizan aproximación de funciones de valor (*Deep Learning*). De ahí la gran diferencia en tiempos ya que, aunque ambos realizan 1000 episodios de entrenamiento, utilizar una red neuronal es más lento (además de que el enfoque tabular no tiene el problema del espacio de acción variable en función del estado y que obtiene las acciones específicas en cada estado en lugar de tener que generalizar).

También conviene mencionar que una de las funcionalidades destacables del algoritmo de *Q-routing* es que es capaz de encontrar varias rutas si son igual de óptimas [20]. En efecto, si modificamos el coste del enlace entre los nodos 4 y 9 de la Figura 6, y le asignamos un valor de 18, el algoritmo saca como resultado ambos caminos óptimos (1,4,9 y 1,4,5,9). Esta funcionalidad no la tiene ni el algoritmo de Dijkstra que nosotros implementamos con la librería Dijkstar, ni nuestros propios algoritmos. Hay que recordar que nuestros algoritmos implementados con ChainerRL utilizan aprendizaje profundo para deducir las mejores acciones en cada estado. Esto implica que acabarán decantándose por una cierta ruta concreta. El ejemplo de *Q-routing* por otra parte, utiliza una metodología tabular, hallando los costes de las mejores rutas. Después de construir la tabla, puede saber las mejores rutas con sus costes, los cuales se pueden comparar y si dos rutas tuvieran el mismo mejor coste, se devolverían ambas.

3.5. Conclusiones

Está claro que nuestra solución es muy inferior en tiempo de entrenamiento (si bien esto es mejorable) por lo que claramente no es un competidor serio. Como ya hemos mencionado, el objetivo era realmente familiarizarse con el aprendizaje por refuerzo y el entorno de trabajo que proporcionan ChainerRL y Gym. Sin embargo, nuestros agentes son capaces de resolver con éxito la red, y eso es sin realmente ajustar de forma minuciosa todos los distintos parámetros de los que disponen y sin utilizar los algoritmos de RL mejor enfocados a este problema. Es por ello por lo que podemos concluir que nuestro principal objeto de estudio tiene altas posibilidades de poder ser resuelto con ChainerRL y Gym, cuestión que abordamos en el siguiente capítulo.

4. Establecimiento dinámico de conexiones en redes de transporte

Ahora que ya nos hemos familiarizado con las herramientas que emplearemos para resolver nuestro problema principal, vamos a analizarlo en detalle. El objetivo de este capítulo, y de esta memoria, es proponer una solución al establecimiento dinámico de conexiones (circuitos) en redes de transporte basado en aprendizaje por refuerzo y teniendo en cuenta la disponibilidad de recursos.

Queremos un sistema que sea capaz de elegir, entre una selección de rutas precalculadas, la mejor en cada estado de la red para establecer un circuito bidireccional solicitado entre un par de nodos origen-destino. Esto se puede hacer con algoritmos tradicionales como los basados en la ruta más corta, por lo que usaremos dos de estos como alternativas comparativas. El primero será un sencillo algoritmo de la ruta más corta. Siempre elegirá enrutar los circuitos por la ruta más corta de entre las precalculadas. Es posible que varias de las primeras rutas calculadas tengan la misma longitud así que, en estos casos elegirá siempre la primera. El segundo algoritmo que usaremos en las comparativas será el algoritmo de las K rutas más cortas (K-*shortest paths*) [23]. Es similar al anterior, pero en lugar de tomar como decisión siempre la primera ruta, elegirá la ruta más corta que cumpla con los requisitos para establecer exitosamente la conexión.

4.1. Entorno

4.1.1. Ideas generales

Al igual que en la sección anterior, volveremos a crear entornos de Gym personalizados. En este caso representarán a la red en la que se estará realizando el enrutamiento de circuitos. Estos entornos recibirán las acciones de los agentes RL y en base a los valores de estas acciones asignarán los recursos necesarios en los enlaces que pertenezcan a la ruta seleccionada. En el caso de que no se disponga de los recursos requeridos para la acción seleccionada, la petición de conexión se considerará denegada. Estaríamos por tanto ante una situación de bloqueo. Una conexión establecida exitosamente conllevará un decremento de las capacidades disponibles en los enlaces de la red y una recompensa positiva, mientras que un bloqueo no afectará a las capacidades disponibles, pero supondrá una recompensa nula (o incluso negativa). Conviene mencionar aquí nuestra forma de precalcular las rutas. El proceso se realiza por medio del algoritmo de las K rutas más cortas (K-*shortest paths*) [23], propuesto en la API de la librería NetworkX de Python [24]. El resultado es una lista con tantas rutas precalculadas como se indiquen, ordenadas de más cortas a más largas. Como en este caso se establecerán y liberarán conexiones a lo largo del tiempo, pues la red estará funcionando de forma continua, desde el punto de vista del aprendizaje por refuerzo no habrá ningún estado terminal. Esto implica que tenemos una tarea continua.

Ahora que estamos hablando del funcionamiento de nuestros entornos, expliquémoslo más a fondo. Como ya hemos dicho, nuestro entorno de Gym representará una red, con su topología concreta, la cual estará formada por N nodos (numerados desde $n = 0$ hasta $N - 1$) y E enlaces bidireccionales ($e = 0$ hasta $E - 1$). También mantendrá el estado sobre la capacidad remanente de los enlaces disponibles en la red, C_e. Esta capacidad tendrá unos valores iniciales que se irán decrementando a medida que se establezcan los circuitos.

En J. Suárez-Varela *et al*. [1] el problema planteado es similar, pero en nuestro caso hay algo de complejidad añadida. Esto es porque los circuitos a establecer no tienen una duración infinita, sino que serán desconectados después de un cierto tiempo, T_l, siendo l el identificador del circuito bidireccional, al contrario de lo que ocurre en el trabajo anterior, en el que se considera una situación de tráfico incremental en la que se establecen circuitos hasta que se bloquea una petición. En nuestro caso abordamos un sistema dinámico de establecimiento y liberación de circuitos. El tiempo que estará establecido una conexión es desconocido para los agentes, y es independiente entre diferentes conexiones. Nuestro entorno permite establecer diferentes distribuciones de estos tiempos, y también hacer que los tiempos medios de duración de las conexiones sean diferentes para distintos nodos de la red (es decir, potencialmente puede haber distintas cargas de tráfico entre distintos nodos). A mayores, otro parámetro también independiente para cada nodo de la red será la tasa a la que se generan las peticiones de establecimiento de circuitos en cada nodo (λ_n). También se puede variar de la misma forma la tasa de transmisión de datos y por tanto la cantidad de recursos que consume un circuito (R_l) y que, en caso de establecerse con éxito, habrá que sustraer de la capacidad remanente (C_e) de cada uno de los enlaces por los que haya sido enrutada dicha conexión. Todo esto permite distribuir la carga de la red de la manera deseada para observar los resultados en multitud de condiciones distintas. Además, es posible realizar simulaciones en las que la distribución de la carga en la red varíe con el tiempo.

Supondremos que los circuitos son bidireccionales (al igual que los enlaces). Esto significa que sin importar cual sea el nodo origen y cual el destino en una petición, los recursos consumidos en los enlaces por la ruta seleccionada serán equivalentes en ambos sentidos. Por simplificar, supondremos que la duración media de todas las conexiones es la misma, T. Definiremos la carga de tráfico normalizada de la red como se indica a continuación en la Ecuación 1:

$$\rho = \frac{\sum_{n=0}^{N-1} \lambda_n T}{N(N-1)/2}$$

Ecuación 1

La Ecuación 1 no considera que distintos circuitos pueden llevar distinto tráfico. Simplemente proporciona la carga en Erlangs, esto es, cuántas conexiones hay establecidas de media en la red, normalizada por el número de conexiones que habría si todos los nodos estuvieran conectados entre sí. Suponiendo una situación de tráfico uniforme, en la que todos los nodos generan la misma carga de tráfico (λ). La carga de tráfico de la red normalizada será:

$$\rho = \frac{2\lambda T}{N-1}$$

Ecuación 2

En nuestros experimentos queremos obtener datos para distintos valores de ρ, y como ya hemos decidido fijar T (al que asignaremos un valor de una unidad de tiempo), podemos sencillamente despejar λ e ir obteniendo la tasa a la que los nodos deben generar peticiones para alcanzar de media esa carga en la red.

$$\lambda = \frac{\rho(N-1)}{2T}$$

Ecuación 3

Antes hemos comentado que nuestra definición ignora qué capacidad piden los circuitos en las peticiones. Aquí conviene comentar la definición de ODU ("*Optical channel Data Unit*"), que será la unidad de medida que usaremos para la capacidad de los enlaces de la red, así como de la capacidad requerida por las peticiones de establecimiento de circuitos. Nos apoyamos en la recomendación G.709 de la ITU [25], donde se define este concepto. Para nuestro análisis es importante entender lo que son los ODUk ("*Optical channel Data Unit*-k"), ya que usaremos varios. La unidad básica es el ODU0, que nosotros usaremos como nuestra unidad de referencia (asumiremos que es equivalente a una unidad de ancho de banda). A partir de ahí, ODU1, ODU2, ODU3 y ODU4 suponen respectivamente 2, 8, 32 y 64 veces la capacidad del ODU0.

Vamos a aprovechar esto para plantear dos casos distintos. En el primero y más sencillo, todos los circuitos consumirán una sola unidad de ancho de banda (ODU0). En el segundo caso nos acercaremos más al entorno planteado en J. Suárez-Varela *et al*. [1] donde utilizan el estándar de ODUk para definir cargas que consumen en unidades de ancho de banda 1, 2, 8, 32 y 64 para ODU0, ODU1, ODU2, ODU3, ODU4, respectivamente. Tal y como plantean en J. Suárez-Varela *et al*. [1] distribuiremos uniformemente las probabilidades de cuál es la capacidad pedida por la petición. En ambos casos supondremos que los enlaces disponen de una cierta cantidad de unidades de ancho de banda expresada en términos del número de ODU0 que soportan.

4.1.2. Observación y recompensa

Recapitulando lo que acabamos de explicar, podemos crear entornos que almacenan el estado de una red en cada momento a medida que se atienden peticiones de establecimiento de conexiones entre los distintos nodos que la conforman. Los distintos nodos serán a su vez fuentes de tráfico que pueden no tener los mismos parámetros. Solamente nos queda describir cómo exactamente interactúan los agentes con estos entornos. Pues bien, como ya hemos dicho, las acciones que estos agentes toman deciden qué rutas precalculadas usar para establecer un circuito, el cual será desconectado un cierto tiempo después. ¿Pero cómo deciden los agentes qué acciones tomar? La respuesta está obviamente en la observación devuelta por el entorno al agente. Esta contiene un elemento por cada nodo existente en la red (llamemos a esto el primer vector de la observación), otro elemento por cada enlace (llamemos a esto el segundo vector de la observación), y un último valor que indica la carga de la petición.

En cada observación, dos de los valores del primer vector serán 1 (los nodos origen y destino de la conexión) mientras que los demás serán 0 (el resto de los nodos). Nótese que puesto que las conexiones son bidireccionales, realmente los dos nodos etiquetados con 1 son tanto origen como destino de la conexión. Trabajar con conexiones bidireccionales simplifica el precálculo de las rutas, ya que calculando las rutas desde un nodo origen hasta otro destino, esas mismas rutas se pueden reutilizar si se invierten el sentido de la petición. Esto último enlaza con nuestra suposición de que los circuitos establecidos son bidireccionales. Otra razón es que simplificar la observación al agente favorece su aprendizaje. Además, la suposición de que no importa qué nodo es el que ha generado la petición es aceptable ya que estamos centrándonos en la carga en la red.

Continuando con la explicación de la observación devuelta por el entorno, pasemos al segundo vector. Estos valores variarán entre 0 y 1, e indican cuanta capacidad le queda a cada enlace, es decir, cuántos recursos son aún utilizables en ese enlace. Como es de esperar, la razón por la que el valor máximo de estos valores es 1 se debe a que están normalizados. Todos se dividen por la capacidad de enlace de máxima capacidad. Esto significa que no todos los enlaces tienen que mostrar un valor de 1 si están sin utilizar. Esto es importante

para indicar a los agentes la diferencia entre capacidades entre distintos enlaces, ya que estos no tienen por qué tener la misma capacidad máxima.

Para terminar con la observación mencionamos el último elemento incluido, la carga de la petición del circuito también normalizada de igual forma que las capacidades de los enlaces. Se espera que este último valor pueda ser usado por los agentes para saber si una ruta tendrá los recursos necesarios antes de tomar una acción, ya que es directamente comparable con los valores mostrados en el segundo vector. También podrá ser utilizado para reconocer distintos ODUk en cada petición, útil si alguno fuera prioritario.

Cada observación indica el estado actual de la red en cuanto a capacidades remanentes en el momento en el que se va a atender una petición y también muestra información sobre esta petición en forma de los nodos a conectar y la carga que este circuito supondrá. Esperamos pues que los agentes sean capaces de aprender cómo afectan las distintas acciones a la red en función de los nodos que se indican en la petición (en el primer vector), y elijan las rutas más adecuadas una vez sepan cuáles son.

La forma en la que los agentes saben si están teniendo éxito o no sigue siendo mediante la recompensa, que sencillamente indica si el circuito se ha establecido exitosamente. Nosotros nos decantamos por dar una recompensa positiva (de valor la carga de la petición) cuando el circuito se ha establecido correctamente, y una recompensa nula, de valor 0, cuando no es así (aunque también hemos experimentado con una penalización). En el caso de que todas las peticiones sean iguales en cuanto a capacidad demandada, esto se traduce en establecer el máximo número de conexiones o lo que es lo mismo, minimizar la probabilidad de bloqueo. También se podría optar por dar recompensas mayores en función de distintos parámetros de la petición, por ejemplo, para dar prioridad a las peticiones de ciertos nodos, o penalizar más fuertemente el bloqueo de una conexión.

Otro detalle importante es que incluimos la posibilidad de denegar las conexiones voluntariamente. Esto será una acción a mayores disponible a los agentes, en caso de permitirse. Dependiendo del escenario activaremos esta posibilidad, pero por defecto asumiremos que no es una opción.

4.1.3. Fuente de tráfico

Antes de pasar a hablar de los agentes dediquemos unos momentos a explicar las distintas formas en las que podemos emplear los entornos en cuanto a la fuente de tráfico se refiere. Antes hemos hablado de cómo cada nodo genera peticiones en función de su tasa de generación de peticiones independiente de los demás. También vimos que estos nodos tenían asociadas duraciones para los circuitos que se establecen en sus peticiones y unas cargas. Para manejar estas fuentes de tráfico, que podrían considerarse independientes, creamos un objeto en el entorno a partir de una clase personalizada que contiene los métodos y atributos necesarios para generar estas peticiones.

Esta clase permitirá, a grandes rasgos, que el entorno sea un simulador de eventos discretos que mantiene una cola de eventos, que representará las peticiones que se generarán dentro de un cierto tiempo. Esto significa que estas peticiones no están a la espera de ser atendidas sino de ser generadas. Ya dijimos que se asume que todas las peticiones se atienden cuando un agente recibe la observación, y este sería también el momento real en el que se habrían generado. Esta cola de peticiones almacena una petición desde cada nodo, y están ordenadas por sus tiempos de llegada correspondientes. Estos tiempos tenderán a ser menores cuanto mayor sea la tasa de generación de peticiones del nodo. Cada vez que el entorno recibe una decisión sobre una petición, realiza las asignaciones adecuadas de recursos (si se conecta con éxito) y salta al siguiente evento (llegada de la siguiente

petición). Al saltar al siguiente evento de petición es posible que en el tiempo de por medio se produzca la desconexión de uno o varios circuitos. Esto son también eventos que se tienen en cuenta, pero no se almacenan como tal los tiempos en los que suceden. En lugar de eso, el salto al siguiente evento de petición involucra la comprobación de si se producen eventos de desconexión en el tiempo transcurrido (se sabe porque las peticiones tienen asociados sus tiempos de duración), y si es así, se liberan los recursos correspondientes. Al sacar una petición de la cola, el nodo cuya petición acaba de ser atendida seguirá generando peticiones, por lo que se vuelve a generar otra petición de este nodo y se introduce en la posición adecuada de la cola. Básicamente tenemos peticiones con una cierta distribución (nos decantamos por una distribución exponencial) asociadas a un nodo, y las peticiones de todos los nodos se superponen ordenadas por sus tiempos de llegada. Se podría decir que la cola descrita almacena las futuras peticiones que serán generadas, pero ya tiene todos sus datos, así como sus tiempos de llegada. Esto quiere decir que siempre hay una petición de cada nodo precalculada en la cola.

Para implementar el mecanismo (simulador de eventos) descrito no se ha usado ninguna librería debido a que no era necesario por su simplicidad. En resumen, los únicos eventos son la llegada de una petición y la desconexión de un circuito. La llegada de una petición involucra una acción por parte del agente, mientras que una desconexión siempre se realizará entre medias de las tomas de acciones. Entre la llegada de una petición y otra pueden suceder varias desconexiones. Estos eventos suponen una liberación de recursos en los enlaces y afectan a la siguiente observación devuelta por el entorno.

4.2. Agentes RL

En cuanto a los agentes RL estos no tienen ningún problema a la hora de interactuar con el entorno, ya que a fin de cuentas el entorno de Gym creado tiene los métodos fundamentales para realizar el entrenamiento. Solamente hay que crearlos igual que lo hicimos en la sección anterior.

Emplearemos los algoritmos que ya hemos detallado brevemente en la sección 2.2.2. La mayoría de los parámetros en estos algoritmos se dejarán por defecto, variando solamente las funciones Q o políticas para que implementen las redes neuronales. Estas redes neuronales, al igual que en el capítulo 3, tendrán tantos nodos de entrada como elementos tenga el vector de observación, con una capa oculta de 60 neuronas y tantas salidas como acciones haya disponibles. Hemos realizado varias pruebas con diferentes cantidades de neuronas en la capa oculta, así como probado añadir otra capa oculta a mayores, pero los resultados no mejoran notablemente. Para el algoritmo TRPO, tanto la política como la función de valor del GAE se implementarán con esta estructura. Las conexiones entre capas siguen siendo lineales con cada nodo teniendo un enlace con todos los nodos de las capas adyacentes y función de activación la tangente hiperbólica.

En todos los experimentos mostrados a continuación los hiperparámetros de los algoritmos se dejaron con su valor por defecto, salvo unas excepciones. El intervalo de actualización del modelo, y (salvo para el TRPO) el tamaño inicial del buffer de *experience replay* deberán ser ajustados adecuadamente y no tendrán el valor por defecto. Realizando varias pruebas con valores diferentes (análisis similar al realizado en Suárez-Varela *et al.* [1]) se llega a la conclusión de que un valor del intervalo de actualización de 50000 es adecuado, ya que es lo suficientemente grande para tener buena precisión en las actualizaciones y no tan grande como para ralentizar el entrenamiento. En cuanto al tamaño inicial del buffer de *experience replay*, se le asigna un valor de 100000 por las mismas razones que el intervalo de actualización (ya que estos parámetros se probaron en conjunto). Valores inferiores a estos

no mejoraban los resultados y acababan por empeorarlos, mientras que los superiores solamente suponían tiempos de entrenamiento mayores.

Los últimos hiperparámetros que modificaremos serán los ya vistos en el capítulo 3. Estableceremos un valor del factor de descuento (γ) de 0.995 ya que tiene sentido que en este escenario deba ser un valor elevado, pero no 1 (es una tarea continua, tenemos entrenamientos largos y un entorno con muchos estados). Probando valores más bajos (0.8 y 0.9) no se obtuvieron mejores resultados. Otra posibilidad es utilizar un sistema miope y establecer un valor de γ de 0, pero esto no mejora significativamente los resultados (en general son peores, dependiendo del algoritmo). El último valor que probamos es 0.5 ya que es intermedio, pero también empeoraba los resultados. La probabilidad de exploración la fijaremos (salvo para el TRPO) a una probabilidad constante de 0.2. Probando con valores de 0.1 y 0.3 no se obtuvieron mejores en el rendimiento. Tampoco se mejoraron los resultados con probabilidades de exploración lineales decadentes que comenzaban en valor de 0.4 y 0.2 y descendían a 0.05 en 500000 pasos.

4.3. Implementación de algoritmos tradicionales usados como comparativa

Ya hemos visto todo el funcionamiento básico de nuestros entornos y agentes RL por lo que pasemos ahora a realizar algunas consideraciones sobre los algoritmos tradicionales que usaremos como comparativa y que interactuarán con los entornos de la misma forma que los agentes RL. De este modo, se puede reutilizar el simulador desarrollado y realizar una comparativa de los distintos algoritmos en las mismas condiciones. Cuando antes hemos hablado de soluciones tradicionales ya comentamos que nos referíamos a algoritmos como los que se basan en elegir las rutas más cortas. En nuestro caso emplearemos como algoritmos alternativos de comparación los dos mencionados, el uso de la ruta más corta y las K rutas más cortas.

La implementación planteada para estos algoritmos es aprovechar que en nuestros entornos las rutas precalculadas están ordenadas de más corta a más larga, cosa que viene de la implementación propuesta del cálculo de estas rutas [24]. Lo único que hay que hacer entonces es crear imitaciones de agentes de ChainerRL que devuelvan siempre la primera acción disponible (o tantas como rutas precalculadas haya para el algoritmo de las K rutas más cortas). Esto siempre le indicará al entorno que se desea usar la ruta más corta. Decimos que son imitaciones porque, aunque implementan los métodos básicos que usaremos en los agentes RL para hacer que actúen y entrenen en el entorno implementado, estos métodos no realizarán los cálculos previstos e ignorarán la mayor parte de los argumentos de entrada. La única razón de usar esta implementación es que los algoritmos alternativos usen exactamente el mismo entorno que los agentes RL.

Hagamos unas consideraciones en cuanto a escoger la ruta más corta disponible (con suficientes recursos) de todas las precalculadas. Este proceso está incluido en los cálculos del entorno implementado, y la imitación del agente de ChainerRL para este algoritmo aprovecha esto. Esto quiere decir que devolverá un vector de tantos elementos como rutas precalculadas haya (K) con valores enteros ascendentes comenzando por 0. Es decir, si tenemos cuatro rutas precalculadas entre las que elegir, este agente devolverá siempre como acción el vector [0, 1, 2, 3]. Esto no es típico ya que los agentes de ChainerRL reales solamente devolverán un valor por cada acción, pero es una excepción manejable y tenida en cuenta en el entorno de Gym.

4.4. Experimentos

Pasemos ahora a detallar los experimentos que se realizarán. Como red de prueba emplearemos la indicada en el artículo [26], la red NSFNet. Su topología puede verse en la Figura 11. La longitud de los enlaces es un parámetro que nosotros ignoramos en esta memoria.

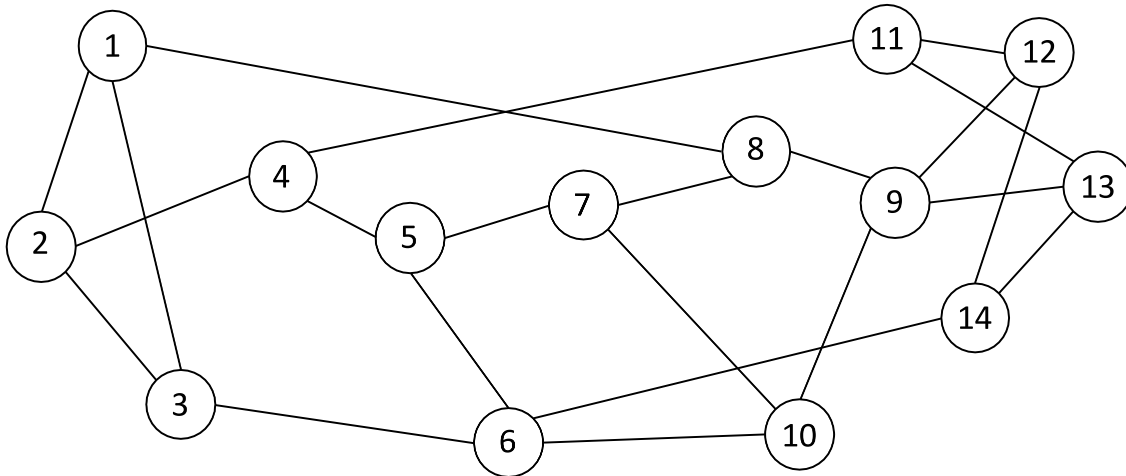

Figura 11 – Topología de la red NSFNet

Vamos a realizar experimentos en los que nos centraremos en el estudio de la probabilidad de bloqueo en estas redes, y en concreto, vamos a observar su evolución a medida que la carga en la red cambia. Describiremos este experimento en el apartado 4.4.1.

En cuanto a la cantidad de rutas precalculadas que empleamos, nos decidimos por 4. Este es el mismo valor que razonaron en Suárez-Varela *et al*. [1] y en nuestras propias pruebas confirmamos las mismas conclusiones. Precalcular menos o más rutas no mejora los resultados.

La metodología seguida al realizar estos experimentos consiste en entrenar a los agentes con 10^6 de peticiones, para después probarlos en una fase de testeo en la que también deberán enrutar 10^6 peticiones. En ambas fases se añaden 1000 peticiones que se consideran de inicialización de la red (para asegurar un estado estacionario), y que no serán usadas para obtener datos. Tanto durante el entrenamiento como el testeo se obtienen métricas sobre el rendimiento de los algoritmos las cuáles usaremos para presentar datos en los próximos experimentos, habiendo usado para las gráficas una ventana de promediado de 10000 peticiones. Las dos fases de entrenamiento y posterior testeo se repiten para cada valor de carga normalizada de la red, es decir, los resultados obtenidos para distintos valores de ρ se obtienen de agentes entrenados específicamente para ese valor de carga y no se testean los agentes entrenados a una carga concreta en otras distintas.

4.4.1. Circuitos transportando tráfico homogéneo con cargas iguales y con carga de red distribuida uniformemente

En este experimento vamos a lanzar unas simulaciones en las que todos los nodos de la red generan tráfico uniforme con las mismas características. El tráfico serán peticiones de establecimiento de un circuito que transportará 1 ODU0 con otros nodos de la red elegidos aleatoriamente con distribución uniforme. También inicialmente todos los enlaces tendrán la misma capacidad de valor 4 ODU0. Estas simulaciones se repetirán para distintos valores de carga, aunque siempre se mantendrán uniformes. Como todos los circuitos son iguales

en cuanto al tráfico que transportan, nos interesa observar que probabilidades de bloqueo obtienen los distintos algoritmos. Los resultados de la Figura 12 muestran los resultados de la fase de testeo en estas condiciones.

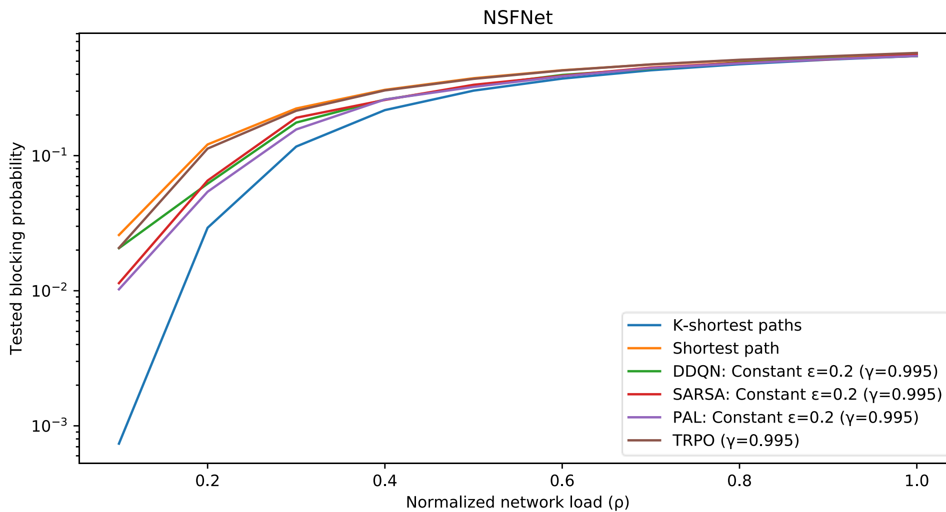

Figura 12 – Probabilidades de bloqueo durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico homogéneo.

Como se puede observar, los algoritmos de la ruta más corta y de las K rutas más cortas definen unos límites entre los que los algoritmos de RL operan. Obviamente los mejores resultados los obtiene el algoritmo de las K rutas más cortas al poder comprobar todas esas rutas en el momento de recibir una petición, y seleccionar la más corta con disponibilidad de recursos. Alcanzar éste parece ser muy difícil para los algoritmos de RL ya que se observa que no obtienen probabilidades de bloqueo tan bajas, pero debe tenerse en cuenta que el uso de estos algoritmos se traduce en probar la disponibilidad de recursos en una sola ruta (la que indica el algoritmo), mientras que en el de las K rutas más cortas, si al probar con la primera no hay éxito se prueba con la siguiente y así sucesivamente. Un detalle interesante es que cuanto mayor es la carga en la red, menor es la diferencia entre ellos. Además, con cargas en la red tan altas las probabilidades de bloqueo alcanzan valores bastante superiores al 10%, los cuales no son nada buenos en ningún caso y carecen de valor.

Como se ha mencionado, todos los algoritmos de RL superan al de la ruta más corta, y concretamente el DDQN, el SARSA y el PAL (los basados en *Q-learning*) lo hacen con bastante margen para cargas bajas en la red. Exceptuando al TRPO, que en estas condiciones obtiene resultados muy similares al de la ruta más corta (y por ello no vale la pena usar por su mayor complejidad y tiempo de cómputo), estos algoritmos otorgan mejores resultados sin necesidad de usar un algoritmo como el de las K rutas más cortas.

Antes de finalizar el análisis de los resultados de este experimento veamos otro aspecto de interés. En la Figura 13 vemos la ratio de acciones que toman los agentes de RL que se corresponden con la ruta más corta, durante la fase de testeo.

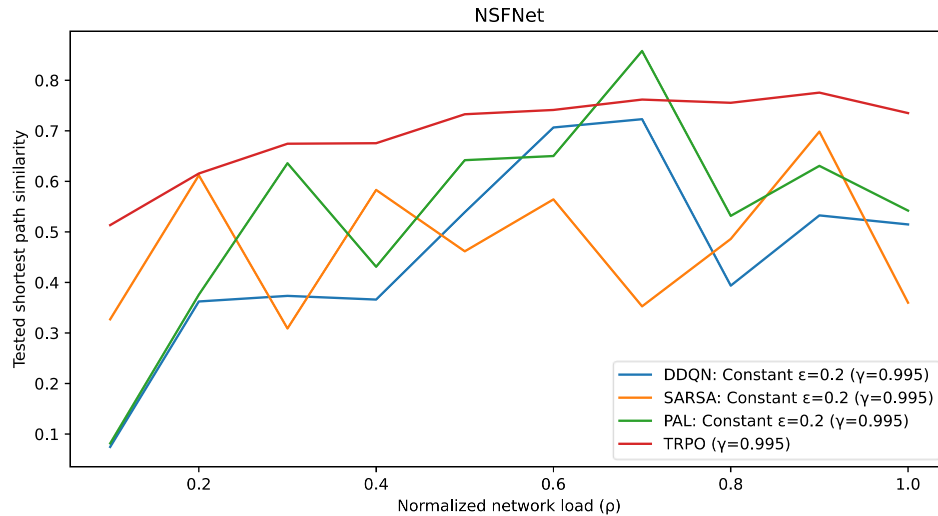

Figura 13 – Probabilidad de que los agentes de RL decidan utilizar la ruta más corta para una petición durante la fase de testeo para la NSFNet con circuitos transportando tráfico homogéneo

Vemos que, en general, los algoritmos suelen decantarse más por la ruta más corta que por las demás (hay 4 rutas por lo que una probabilidad de más del 25% lo indica). Esto es especialmente cierto para el TRPO que se decanta con bastante probabilidad por esta ruta. Esto tiene sentido viendo que este algoritmo es el que obtenía los resultados más similares al algoritmo de la ruta más corta en la Figura 12. Los demás agentes son bastante erráticos en cuanto a estos resultados y la varianza de estas probabilidades es alta. Como solo hemos simulado un solo agente de cada tipo, volviendo a realizar las simulaciones, los resultados para el DDQN, SARSA y PAL en la Figura 13 no serían los mismos (aunque seguirían siendo mayores que el 25% normalmente).

Con todo esto, y especialmente los resultados de la Figura 12, podemos concluir que la utilidad de los algoritmos de RL es prometedora en el ámbito y observamos que los algoritmos basado en *Q-Learning* son mejores que el TRPO. Como esto es una prueba bastante simple comprobemos en el siguiente experimento un escenario que añade más complejidad al problema para comprobar si los resultados se mantienen similares.

4.4.2. Circuitos transportando tráfico heterogéneo y con carga de red distribuida uniformemente

Para este experimento el escenario será similar al del apartado 4.4.1 pero los circuitos no siempre transportarán el mismo tráfico. Como ya dijimos usaremos distintos ODUk, que serán ODU0, ODU1, ODU2, ODU3, ODU4, los cuales equivalen a cargas de 1, 2, 8, 32 y 64 ODU0, respectivamente. Como en este caso los enlaces van a requerir mayor capacidad les asignamos un valor de 200 ODU0 a cada uno, igual que hacen en J. Suárez-Varela *et al.* [1].

Lo primero es decidir qué métrica usaremos en este caso para establecer la comparativa. Como ahora los circuitos no tienen la misma capacidad, sino que varían de forma uniforme entre distintos valores, no nos vale con comprobar la probabilidad de bloqueo. En lugar de eso, vamos a comprobar la cantidad de ODU0s que bloquean en total los algoritmos. Asumiendo que el beneficio obtenido es proporcional a la cantidad de ancho de banda enrutado, esto es adecuado.

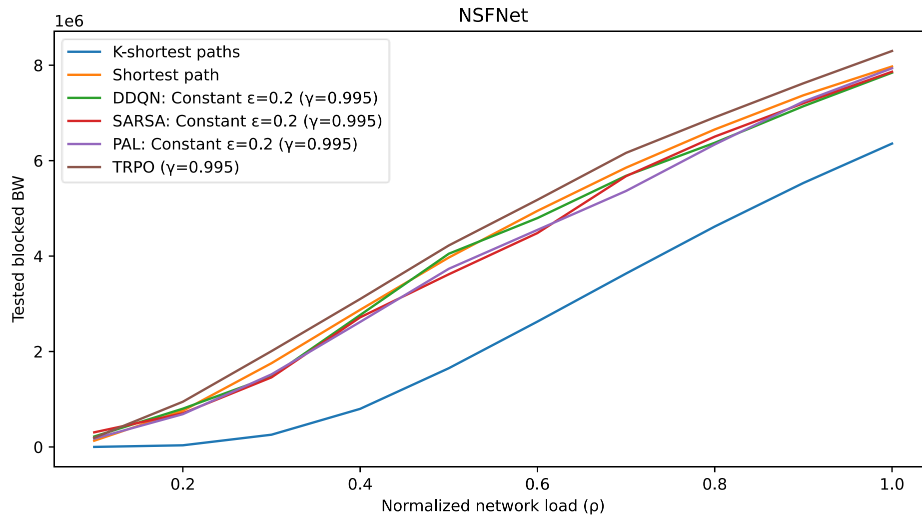

Figura 14 – Ancho de banda total bloqueado por los agentes en ODU0s durante la fase de testeo en función de la carga de la red para la NSFNet con circuitos transportando tráfico heterogéneo

Los resultados que se observan en la Figura 14 se podrían considerar globalmente peores que en el experimento anterior, pero como la métrica no es igual que antes es difícil decir si esta es una conclusión acertada. Como puede observarse los algoritmos de RL tienen un rendimiento muy similar al algoritmo de la ruta más corta salvo para el TRPO que obtiene peores resultados (mientras que antes esto no sucedía). Basándonos en eso, es razonable afirmar que la complejidad añadida en la introducción de circuitos transportando distintas cantidades de tráfico produce un empeoramiento del rendimiento de los algoritmos de RL.

Para confirmar esto, podemos observar en la Figura 15 las probabilidades de bloqueo observadas en este caso, aunque como ya hemos dicho, no es una métrica fiel de los resultados que están obteniendo los algoritmos (solo con enrutar los circuitos de mayor capacidad el resultado sería muy bueno en términos de ancho de banda cursado teniendo probabilidades de bloqueo altas).

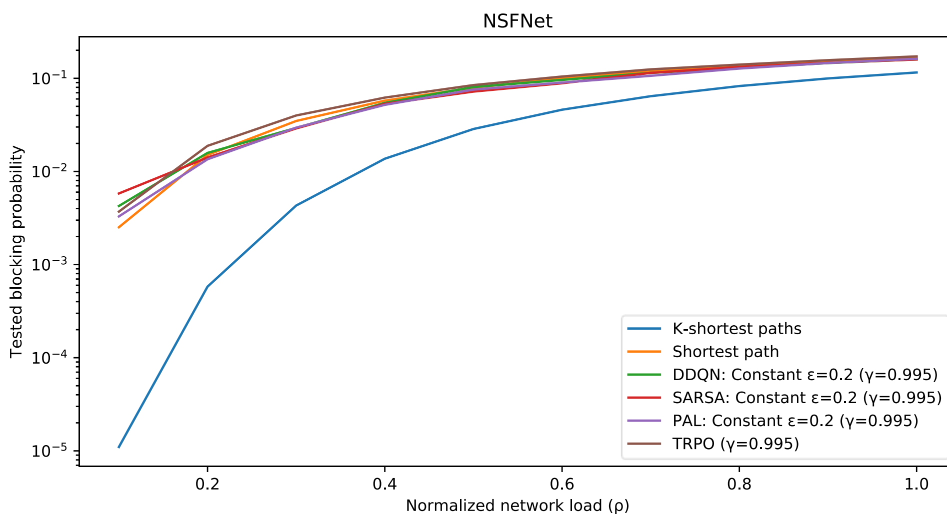

Figura 15 – Probabilidades de bloqueo durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo

Viendo estas probabilidades, nuestras conclusiones parecen razonables. Se ve que las probabilidades de bloqueo de los algoritmos RL son muy similares al algoritmo de la ruta más corta, y no tenemos razones para pensar que esto es debido a que los algoritmos de RL han hallado una solución mejor. Con este último comentario nos referimos a que existe la posibilidad de que los algoritmos estén ignorando circuitos que transporten poco tráfico y enrutando bien los que transporten mucho tráfico, lo cual aumentaría la probabilidad de bloqueo, pero no disminuiría mucho el beneficio). Lo podemos confirmar observando el beneficio que obtienen en la Figura 16.

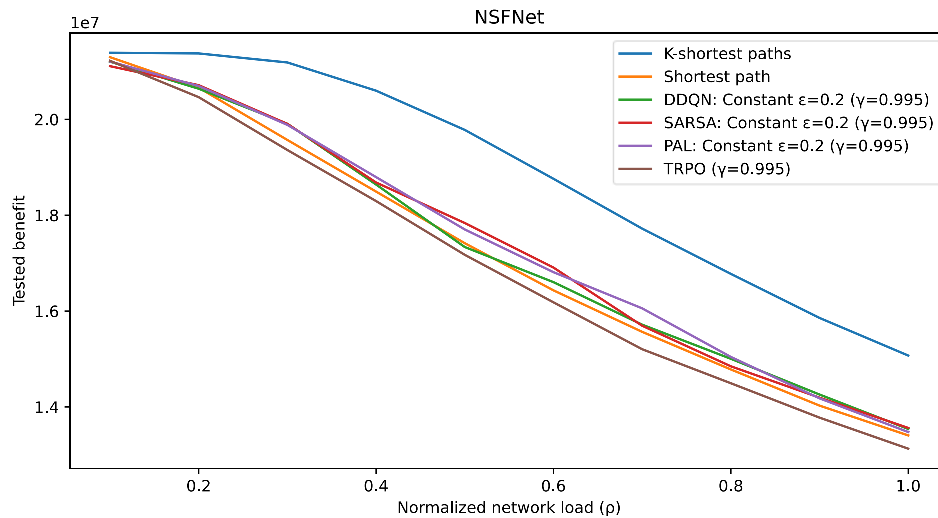

Figura 16 – Beneficio durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo

Observando el beneficio, que como ya hemos dicho es proporcional a la cantidad de ancho de banda que los algoritmos enrutan con éxito, no cabe duda de que en este escenario, el uso de algoritmos de RL no supone una ventaja significativa respecto a un algoritmo mucho más simple y menos costoso como el de la ruta más corta. Aunque los algoritmos DDQN, SARSA y PAL superan al rendimiento de la ruta más corta, la mejora es muy poco significativa.

Aquí planteamos una modificación al experimento consistente en penalizar a los agentes con una recompensa negativa de igual valor al tráfico que lleva cada circuito. Esto quiere decir, que obtendrán un valor positivo o negativo de recompensa proporcional al tráfico que transporta el circuito dependiendo de si enrutan con éxito la conexión o la bloquean. Esto tiene sentido ya que penalizar más el bloqueo de circuitos que lleven más tráfico es lógico ya que la perdida de beneficio es mayor. En la Figura 17 vemos el beneficio obtenido en este caso.

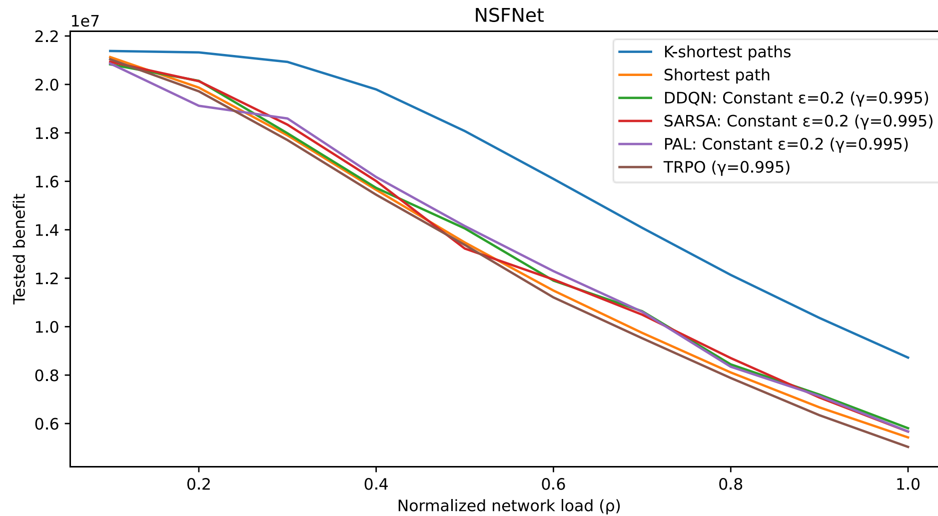

Figura 17 – Beneficio durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo y penalización

Tal como se ve, la Figura 16 y la Figura 17 son prácticamente iguales. Es más, al haber mayor diferencia entre recompensas de conexión y bloqueo esto es de esperar. Con esto concluimos que añadir las penalizaciones no aporta una mejora considerable.

Sí que es cierto que aunque no tiene mucho sentido usar RL de esta forma en este escenario, existe la posibilidad de que los algoritmos estén funcionando correctamente y sencillamente la solución óptima al problema es una política de la ruta más corta. Sin embargo, es más probable que este no sea el caso y que el entrenamiento de los agentes es insuficiente. También es razonable pensar que haría falta una red neuronal más avanzada para encontrar una política mejor. Para llegar a esta conclusión nos volvemos a basar en la similitud que presentan las políticas de estos algoritmos a la ruta más corta, que podemos ver en la Figura 18 (donde se observa que, en general, la política que siguen los algoritmos de RL no es la ruta más corta).

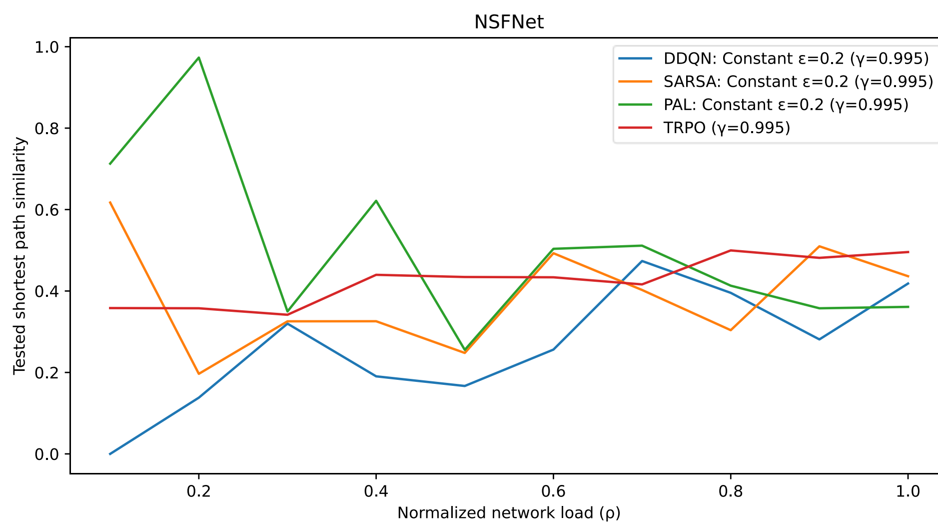

Figura 18 – Probabilidad de que los agentes de RL decidan utilizar la ruta más corta para una petición durante la fase de testeo para la NSFNet con circuitos transportando tráfico heterogéneo y sin penalización

Ahora queda plantearse si no existe algún escenario en el que sí sea ventajoso el uso del aprendizaje por refuerzo. Para ver esto pasamos a ver unos experimentos que introducen un control de admisión.

4.4.3. Circuitos transportando tráfico heterogéneo, con carga de red distribuida uniformemente y control de admisión

Ampliando el escenario del experimento del apartado 4.4.2 pasamos a nuestro siguiente experimento. La única diferencia con el caso anterior será que ahora los agentes recibirán una recompensa de 0.001 unidades por todas las conexiones que no pidan más de 50 ODU0, es decir, por las peticiones que involucren un ODU0, ODU1, ODU2 u ODU3. Esto significa que los únicos circuitos que valdrá la pena enrutar serán los que requieran un ODU4 = 64 ODU0, cuya recompensa seguirá siendo proporcional a su carga (concretamente la recompensa será 64 unidades). Para que esto tenga sentido, ahora se les debe permitir a los agentes denegar conexiones voluntariamente, por lo que disponen de esta acción (es decir, se incorpora un sistema de control de admisión, pero sin decir explícitamente qué conexiones deben denegar ni cuándo hacerlo, sino que tendrán que aprenderlo).

En este escenario, además de comparar con el algoritmo de la ruta más corta y de las K rutas más cortas (sin control de admisión), también compararemos con una versión de dichos algoritmos que sí incorpore control de admisión, indicada en este caso de forma explícita: solamente se enrutarán por la ruta más corta los circuitos que cumplen el requisito del control de admisión entendiéndose como tales a aquellos circuitos que pidan una carga correspondiente al ODU4, y se denegarán voluntariamente los demás. De todas formas, debe notarse que esta política no es óptima en términos de beneficio, pues si la red está muy descargada podrían admitirse esas conexiones y así recibir el beneficio que dan, pues realmente no impactan en términos de aumentar los bloqueos en la red. A estas variantes de los algoritmos las denominaremos, indistintamente, algoritmos de ruta más corta prioritaria o algoritmos de ruta más corta con control de admisión. Respecto a la implementación de estos algoritmos como agentes en el entorno de simulación, además de enrutar por la ruta más corta, en caso de no cumplirse el control de admisión devuelven la acción correspondiente a la denegación de la conexión. Para el algoritmo prioritario de K rutas más cortas, como este devuelve un vector para probar todas las rutas de más corta a más larga, ahora este vector será la acción tomada si el control de admisión se cumple, y de no ser así, devolverá la acción de denegación.

Al igual que anteriormente, debemos elegir una métrica que inequívocamente compare lo buenos que son los algoritmos entre sí. Usaremos para esto el beneficio que obtienen.

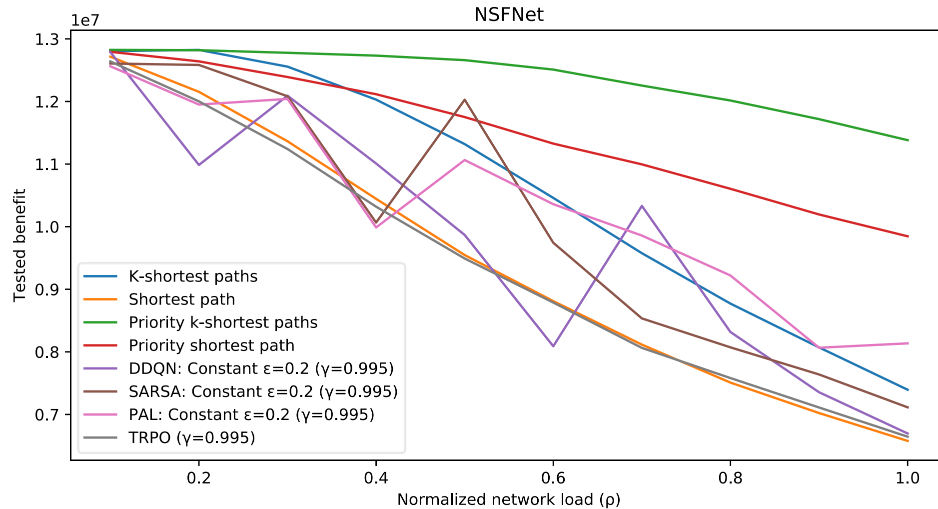

Figura 19 – Beneficio durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo y control de admisión que devalúa los circuitos que transportan poco tráfico

En la Figura 19 vemos cómo ahora el óptimo lo alcanza el algoritmo de las K rutas más cortas con control de admisión. La razón por la que ocurre esto es obvio, enrutar circuitos que apenas dan beneficio pero que consumen recursos en la red no es una buena estrategia (sobre todo en cargas altas). Es por esto por lo que mientras que los algoritmos de la ruta más corta anteriores (sin control de admisión) se quedan estancados, los algoritmos de RL no lo hacen al ser capaces de adaptarse a esta situación aprendiendo a realizar control de admisión (sin estar programada la política de admisión de forma explícita). Sin embargo, es muy aparente que la varianza en los resultados de los algoritmos de RL es muy alta, tal y como indican los resultados tan dispares de sus curvas.

Con solamente estos datos, no se puede afirmar con fiabilidad cómo se compara el rendimiento de los algoritmos de RL con los métodos tradicionales, por lo que debemos reducir esta varianza. El problema es que simular réplicas de DDQN, SARSA, PAL y TRPO es muy costoso, así que optamos por una solución diferente. Vamos a modificar el control de admisión para que sea más obvio cual es la solución óptima. Actualmente, enrutar un circuito que transporte poco tráfico técnicamente no es malo. Si bien otorga poca recompensa, esta no es nula. Incluso si lo fuera, los resultados no cambiarían. El problema es que no es obvio si enrutar un circuito es una mala idea o no para los agentes. Como enrutar un circuito con poco tráfico no supondrá siempre un bloqueo que tenga un impacto significativo en la recompensa a largo plazo, los algoritmos no se deciden claramente por qué hacer.

La modificación del escenario que soluciona esto es sencillamente penalizar el establecimiento de circuitos que den muy poco beneficio (los que se soliciten para transportar un ODU0, un ODU1, un ODU2 o un ODU3). Con una penalización suficientemente alta (comprobamos que -10 es suficiente), los algoritmos de RL dan lugar a un aprendizaje más efectivo de la política de control de admisión que deben adoptar. En la Figura 20 vemos los resultados.

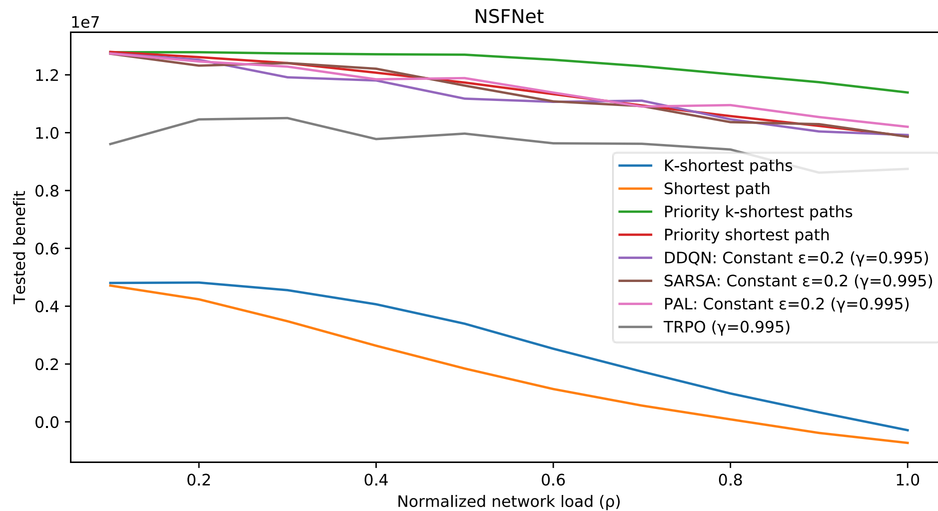

Figura 20 - Beneficio durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo y control de admisión que penaliza los circuitos que transportan poco tráfico

Ahora podemos ver cómo los algoritmos de RL optan por una política de encaminamiento mejor y con mucha menos varianza. Para los resultados del experimento correspondiente a la Figura 19, analizando las probabilidades de bloqueo voluntario que presentaban los algoritmos de RL obteníamos valores bastante dispares y variantes. Por ejemplo, para ρ = 0.5 el algoritmo DDQN denegaba voluntariamente los ODU0, ODU1, ODU2, ODU3 y ODU4 con probabilidades de 0.46%, 0.53%, 11.7%, 69,55% y 0%, respectivamente. Ya comentamos que no denegar las peticiones que cursarán poco tráfico no tenía por qué suponer bloqueos de otras conexiones debido a que estas suponían un impacto pequeño en los recursos de la red. Esto explica por qué las probabilidades de bloquear las peticiones de ODU0s y ODU1s son tan bajas, y se incrementan para ODU2s y ODU3 (ya que estas tienen mucha más probabilidad de suponer un bloqueo de una petición de ODU4, y recordemos que las ODU4 son las únicas que dan un beneficio significativo).

Con todo esto se nota que estas probabilidades favorecen a los circuitos que dan un beneficio significativo, pero no les dan prioridad absoluta (lo cual sería mejor, como el algoritmo de la ruta más corta prioritaria). Además, las probabilidades resultantes, si bien tienen la tendencia mencionada, como ya se ha dicho, tienen mucha varianza y no siempre tenían sentido. En cambio, al penalizar el establecimiento de circuitos que transporten poco tráfico, esto ya no es así. Ahora se observa que los algoritmos de RL deciden bloquear siempre los circuitos que transportan un ODUk distinto a un ODU4 (las probabilidades de bloqueo voluntario para dichos circuitos eran del 100%, y para ODU4s del 0%). Esta es la razón de que ahora sus resultados tengan una varianza mucho más baja, vista en la Figura 20.

Como ahora penalizamos el establecimiento de ciertos circuitos, los algoritmos de la ruta más corta y de las K rutas más cortas sin control de admisión obtienen, lógicamente, resultados peores. Lo realmente interesante es, por tanto, analizar cómo se comparan los algoritmos de RL con las versiones con control de admisión.

Pues bien, vemos que DDQN, SARSA y PAL alcanzan un rendimiento prácticamente igual al del algoritmo de la ruta más corta con control de admisión. Nuevamente, el TRPO no alcanza ese rendimiento y tiene resultados algo peores, pero aun así no son malos. Con todo esto podemos concluir que el aprendizaje por refuerzo es una opción adecuada en el caso de

tener escenarios con factores añadidos a tener en cuenta como, por ejemplo, un control de admisión. Sin embargo, ¿vale la pena usarlos?

Como sus resultados son tan similares al del algoritmo de la ruta más corta prioritaria (esto es, el algoritmo de ruta más corta con control de admisión), lo lógico sería usar este. En efecto, esta es la decisión acertada, pero hay que tener en cuenta si los algoritmos de RL son capaces de alcanzar un resultado aún mejor si fuera posible. Para comprobar esto observamos en la Figura 21 las probabilidades de bloqueo voluntario que en este caso se obtienen. Estas probabilidades son la fracción de bloqueos realizados voluntariamente respecto a los totales.

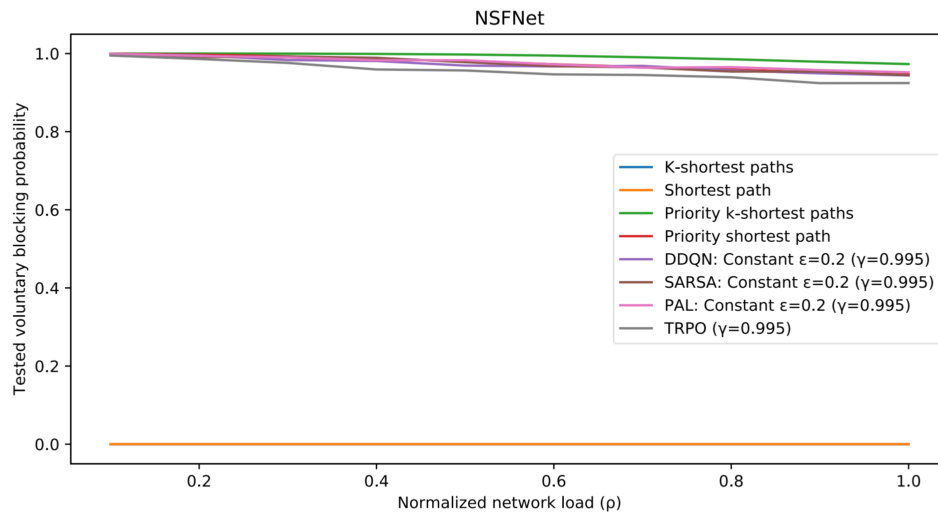

Figura 21 – Probabilidades de bloqueo voluntario durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico heterogéneo y control de admisión que penaliza los circuitos que transportan poco tráfico

Lo que podemos ver es que la probabilidad de bloqueo voluntario de los algoritmos de interés en este caso apenas baja del 100%. Es cierto que viendo los valores exactos para cargas altas observamos que esto ya no es así, observándose aproximadamente un 5% de bloqueos no voluntarios. Estos bloqueos incluyen los de todos los distintos circuitos (independientemente de la cantidad de tráfico que cursen), y, para los circuitos que aportan un beneficio significativo esto supone que aproximadamente un 20% (de estos) son bloqueados. Esto no es mucho y nos indica que para este escenario se ha alcanzado un rendimiento cercano al óptimo. Aun así, la cuestión de si merece la pena usar RL para estos casos sigue abierta. Es cierto que para este escenario no merece la pena, pero es posible que en otros casos sí (por ejemplo, si el algoritmo óptimo no es realizable por alguna razón).

4.4.4. Circuitos transportando tráfico homogéneo, con carga de red distribuida uniformemente y control de admisión

A luz de los resultados del experimento de 4.4.3 vamos a plantear una situación con control de admisión algo distinta para ver si los algoritmos logran mejorar sus resultados. Esta vez volveremos al caso en el que todos los circuitos transportan un ODU0 y la red tiene enlaces con capacidades de 4 ODU0s. Supondremos que el tráfico es uniforme y se genera igual que anteriormente. Sin embargo, a este tráfico se le asignará de forma aleatoria y con distribución uniforme una de tres categorías. Estas categorías indicarán la prioridad de estos circuitos, suponiendo una mayor recompensa (y beneficio) las de mayor prioridad.

Para implementar esto se ha de realizar una pequeña modificación a la observación del entorno. Basta con añadir un elemento a mayores que tomará un valor entre 0 y 1 (valores normalizados) para cada categoría. Esto significa que este valor será 1/3, 2/3 y 1, para las categorías ordenadas de menor prioridad a mayor prioridad.

Con este planteamiento, esperamos que cuanto mayor sea la carga más convenga denegar voluntariamente las conexiones de los circuitos de menor prioridad para poder enrutar los de mayor prioridad. Aun así, denegar todos los circuitos de una menor prioridad no es una buena idea ya que resultará en una pérdida de beneficio.

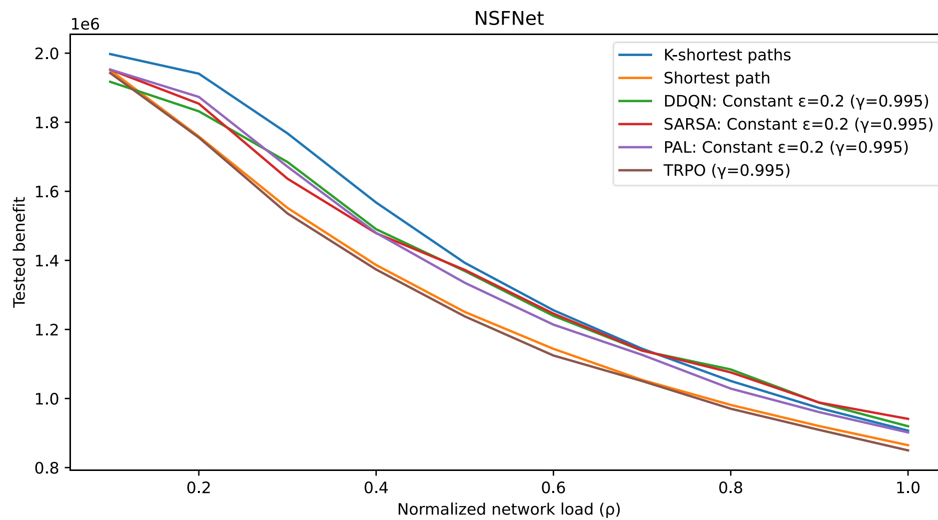

Figura 22 – Beneficio durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico homogéneo y control de admisión que categoriza por prioridades los circuitos

En la Figura 22 podemos ver como los algoritmos de DDQN, SARSA y PAL claramente aprovechan las distintas prioridades de los circuitos para obtener un beneficio mayor que anteriormente (recordemos que, en cuanto a probabilidades de bloqueo, esta situación es idéntica que en 4.4.1). Esto queda claro ya que, para cargas elevadas, donde no se puede esperar no bloquear una cantidad considerable de conexiones, el beneficio de estos algoritmos supera al de las K rutas más cortas.

Está claro que los algoritmos alternativos basados en la ruta más corta no están optimizados para esta situación. Lo ideal sería que estos denegarán una fracción de circuitos en función de su categoría y de la carga de la red, para optimizar el beneficio. Sin embargo, su implementación actual nos sirve al menos para mostrar cómo se adaptan los algoritmos de aprendizaje RL (los basados en *Deep Q-Learning*, ya que el TRPO obtiene resultados prácticamente iguales a la ruta más corta).

Podemos confirmar estas conclusiones viendo las probabilidades de bloqueo voluntario de las distintas categorías de circuitos que presentan los algoritmos de RL a cargas altas. Tomaremos como medida los resultados cuando ρ = 1, ya que para este valor es donde más discriminan los agentes las prioridades. Para valores menores de ρ los resultados son similares pero las probabilidades se reducen de forma generalizada.

	Prioridad baja	Prioridad media	Prioridad alta
DDQN	39.41%	10.32%	2.39%
SARSA	48.76%	7.95%	1.67%
PAL	6.33%	1.89%	0.49%
TRPO	32.65%	2.64%	0.12%

Tabla 12 – Probabilidades de bloqueo voluntario para los algoritmos de RL en una situación de control de admisión por categorías

En la Tabla 12 vemos que por lo general los algoritmos se decantan por denegar con mucha más frecuencia los circuitos de baja prioridad, tal como esperábamos. Otra observación interesante es que el algoritmo PAL, cuyos resultados se asemejan mucho al K rutas más cortas, es con diferencia el que menos circuitos deniega. El hecho de que no aplique demasiado el control de admisión es la razón por la que su beneficio es menor que el del DDQN y SARSA. También se puede comentar sobre la diferencia entre DDQN y SARSA. En la Figura 22 se ve como SARSA obtiene resultados ligeramente mejores para la carga de la red de la Tabla 12. Esto se puede atribuir a que bloquea más circuitos de baja prioridad en lugar de las otras, además de bloquear menos circuitos de media y alta prioridad. Por último, el caso del TRPO es interesante ya que sus resultados eran bastante inferiores a los demás, pero se puede ver que sí aplica control de admisión. Esto significa que el problema en este caso sigue siendo la selección de las rutas.

De estos valores se podría extraer un algoritmo modificado de las K rutas más cortas que mejore el rendimiento del original. Sin embargo, no tiene mucho sentido hacerlo aquí ya que podemos estar prácticamente seguros de que los resultados que obtienen los algoritmos de RL no son óptimos, por lo que estas probabilidades no son las mejores. Para llegar a esta conclusión nos basamos en los resultados de los anteriores experimentos que claramente muestran que las implementaciones de estos algoritmos y del entorno de entrenamiento no son los más adecuados, y se requieren un estudio más profundo. A mayores, aunque bloquear algunos de los circuitos de mayor prioridad puede tener sentido (para no cargar enlaces importantes en algún determinado momento, por ejemplo), lo normal sería que estas probabilidades de bloqueo sean aún más bajas de lo que se ven en la Tabla 12. Aun así, cómo saber que fracción de circuitos de una cierta prioridad se deben denegar para alcanzar el beneficio óptimo depende de muchas variables del estado de la red, y no es fácil de calcular, el aprendizaje por refuerzo es una buena forma de obtener estos valores. De hecho, se podría usar un algoritmo que no implementa RL para enrutar conexiones y uno de RL en paralelo para decidir si se deben enrutar, efectivamente obteniendo las probabilidades de bloqueo voluntario óptimas de forma dinámica.

4.5. Análisis del entrenamiento propuesto y posibles mejoras

En el apartado 4.4 hemos visto los resultados en diferentes situaciones del entrenamiento propuesto. Recordemos que este entrenamiento consistía sencillamente en que los algoritmos enrutaran en la red conexiones cuyos orígenes y destinos eran seleccionados aleatoriamente con distribución uniforme. Vimos que los resultados por lo general no alcanzaban los valores deseados y la razón más obvia de esto era que no aprendían lo suficientemente bien cómo enrutar las conexiones por la red, es decir, el uso adecuado de los enlaces y lo que esto supone. Aunque encontrar una solución a este problema es importante solamente mencionamos como podría abordarse en el apartado 5.2

presentando una línea futura al respecto, mientras que aquí nos centraremos solamente en cómo mejorar el entrenamiento ya implementado.

Un detalle significativo en las pruebas presentadas anteriormente es que los algoritmos eran entrenados para una carga de red concreta, para después funcionar a esa carga. Pero esto no es nada típico en una red real. Los algoritmos deberían poder funcionar en una red mientras la carga cambia. Veamos qué ocurre en la situación del experimento del apartado 4.4.1 cuando los agentes se entrenan a cargas normalizadas de 0.1, 0.5 y 1, para después funcionar a todas las cargas que van desde 0.1 hasta 1. Los resultados se representan en las Figura 23, Figura 24 y Figura 25, respectivamente.

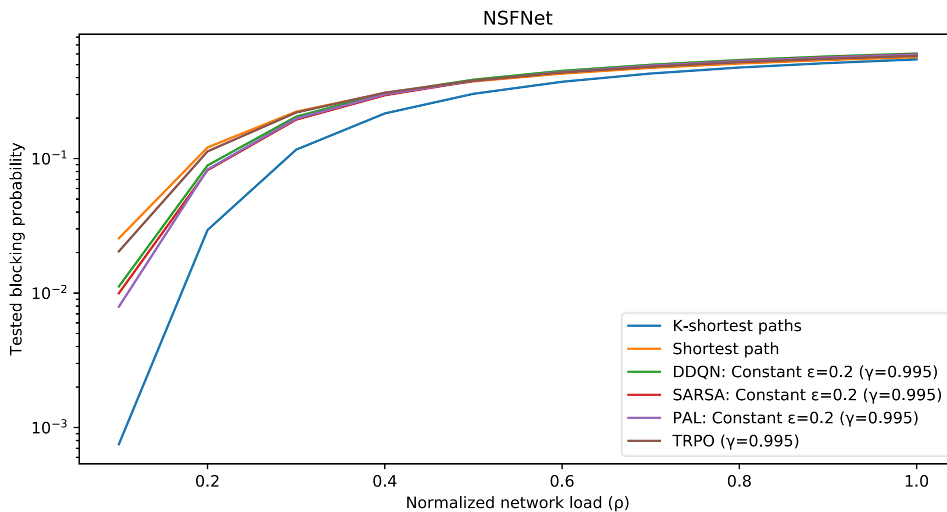

Figura 23 – Probabilidades de bloqueo durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico homogéneo para agentes entrenados en una carga normalizada de 0.1

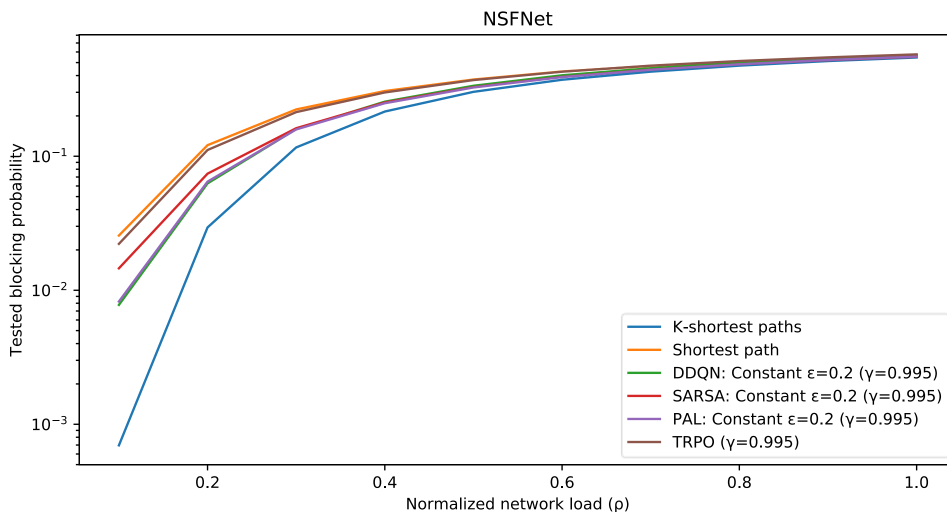

Figura 24 – Probabilidades de bloqueo durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico homogéneo para agentes entrenados en una carga normalizada de 0.5

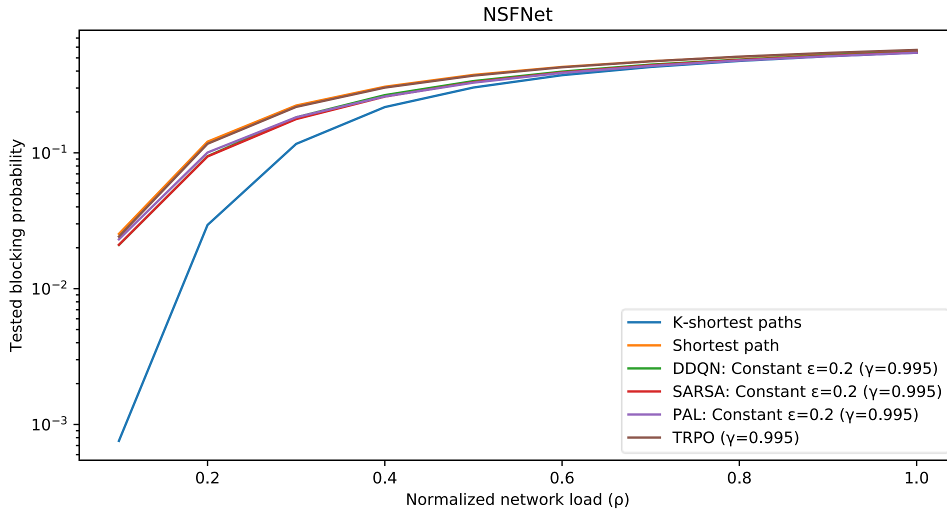

Figura 25 – Probabilidades de bloqueo durante la fase de testeo en función de la carga de la red media para la NSFNet con circuitos transportando tráfico homogéneo para agentes entrenados en una carga normalizada de 1

Como podemos ver, queda claro que la carga a la que se entrenan estos agentes influye en su política. Con los resultados de la Figura 25 podemos afirmar que entrenar los agentes a cargas muy elevadas tiene repercusiones muy negativas en sus actuaciones a cargas bajas. Esto es bastante razonable, ya que si la carga es muy elevada la política óptima consiste en minimizar todo lo posible en consumo de recursos de la red, lo que implica seleccionar las rutas más cortas. De ahí sale también la razón por la que sus probabilidades de bloqueo sean tan similares a la ruta más corta, tal como se puede ver en la Figura 26.

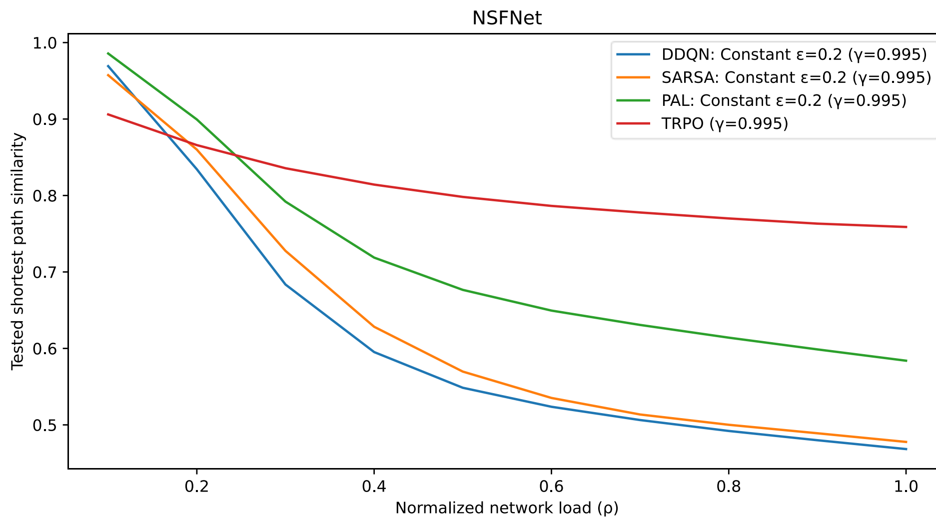

Figura 26 – Probabilidad de que los agentes de RL decidan utilizar la ruta más corta para una petición durante la fase de testeo para la NSFNet con circuitos transportando tráfico homogéneo para agentes entrenados en una carga normalizada de 1

En cuanto a entrenar en cargas bajas, como vemos en la Figura 23 esto tampoco es ideal porque a cargas altas los algoritmos empeoran su rendimiento. Esto no es tan importante porque a estas cargas las probabilidades de bloqueo de todos los algoritmos convergen, pero de todos modos se debería poder mejorar.

Pues bien, tal como se muestra en la Figura 24 un entrenamiento a una carga de red media (0.5) es la mejor opción de las tres. No se observan pérdidas de rendimiento significativas en ningún valor, y de haberlas se pueden atribuir a la varianza que llevamos observando hasta ahora. No podemos afirmar que el valor de 0.5 sea el mejor, pero, en este escenario, uno cercano lo será.

De todas formas, la opción que consideramos más prometedora es entrenar a los agentes utilizando experiencias recopiladas en la red operando a distintas cargas. Esta opción no se ha explorado en este TFG, pero es una línea futura interesante.

4.6. Conclusiones

En este capítulo hemos visto que utilizando aprendizaje por refuerzo profundo se pueden establecer dinámicamente circuitos en una red. Por lo general los algoritmos de RL tal y como los proponemos se mantienen entre dos "cotas" en cuanto a rendimiento. Estas "cotas" son los algoritmos de la ruta más corta y de las K rutas más cortas.

Nuestro entrenamiento y configuración de los agentes es en gran medida básico, y queda claro que esto no es suficiente para obtener resultados superiores al algoritmo de las K rutas más cortas en la mayoría de los escenarios (en gran medida debido a que prueba con varias rutas en lugar de una sola al enrutar). Sin embargo, demostramos como sí superan al algoritmo de la ruta más corta. Esto es significativo debido a que demuestra que de tener que seleccionar una sola ruta por la que encaminar una conexión, escoger siempre la misma no es óptimo, y con RL se puede aprovechar el estado de la red para deducir otras rutas mejores.

Otra conclusión importante a la que llegamos es que los agentes de RL presentan una buena flexibilidad, pudiendo aplicar control de admisión. En las pruebas realizadas se demostraba cómo de haber conexiones que supongan un beneficio reducido, estas tenían menor prioridad al ser enrutadas. Vimos que la forma en la que esto se podía variar, dando los mejores resultados cuando las peticiones tenían una prioridad claramente definida. Un comentario interesante es que observando las probabilidades de que los algoritmos de RL bloqueen voluntariamente conexiones veíamos como estas se enrutaban aun cuando su recompensa era baja, siempre y cuando afectaran poco al enrutamiento de conexiones más importantes. Esto último añade aún más mérito al aprendizaje por refuerzo.

Respecto al problema principal observado que limita el rendimiento del RL, parece que es debido a la escala de la red. Una gran cantidad de nodos y enlaces en la red complican el entorno hasta tal punto que los agentes necesitan un entrenamiento muy exhaustivo, pero realizar entrenamientos muy largos a los agentes es complicado. Estos pierden conocimientos y se suelen estancar en puntos no óptimos, por lo que dejan de presentar mejoras en su rendimiento. De hecho, comprobamos que adiciones como un control de admisión no afectan especialmente a estos resultados y la carga de entrenamiento puede ser promediada. La conclusión principal es que lograr entrenar a los agentes de RL de una forma que les permita aprender a enrutar conexiones eficientemente en la red (aunque sea de forma indirecta) es una cuestión aún por resolver.

Es por ello que, el principal objetivo a conseguir en futuros trabajos sería encontrar un entrenamiento que escale bien con las redes, centrándose en algoritmos basados en el aprendizaje de la función de valor de la acción, Q (ya que el TRPO, basado en gradiente de la política, claramente no era la mejor alternativa).

5. Conclusiones y líneas futuras

5.1. Conclusiones

En este TFG nos propusimos estudiar la utilidad del aprendizaje por refuerzo en el enrutamiento de conexiones en redes telemáticas. Se trataba de un estudio básico de los fundamentos del ámbito con el objetivo de ver la facilidad de uso y rendimiento que tienen estas técnicas en un entorno relacionado con las telecomunicaciones.

Inicialmente planteamos en el capítulo 3 usar aprendizaje por refuerzo profundo, con la ayuda de librerías de código abierto (ChainerRL y Gym), en un problema de la búsqueda de la ruta de menor coste. Comparamos los resultados de utilizar el algoritmo DDQN con una configuración básica con respecto al bien conocido algoritmo de Dijkstra y un algoritmo de aprendizaje por refuerzo tabular. Nuestras observaciones en dos redes distintas con características potencialmente problemáticas para el problema (una bifurcación significativa y un sumidero) demostraban que el uso de aproximación de funciones presente en las redes neuronales aumenta la complejidad de la solución innecesariamente, dando como resultado tiempos de cómputo más lentos. Aun así y como era de esperar, nuestra solución si lograba calcular la ruta menos costosa exitosamente.

En el capítulo 3 no buscábamos realmente comprobar si el aprendizaje por refuerzo profundo sería mejor que las soluciones más simples. Obviamente el algoritmo de Dijkstra daba los mejores resultados ya que las redes analizadas no eran grandes. Además, si bien es cierto que el algoritmo de *Q-routing* analizado fue incapaz de calcular la ruta más corta en una red concreta, mientras que nuestra solución sí, indicando una mayor fiabilidad, es difícil afirmar que esto realmente no era provocado casualmente por un error en esa implementación concreta del mismo. Por ello, aun así, creemos que una solución tabular seguiría siendo una opción más eficiente. Dicho esto, pudimos comprobar que usando ChainerRL en conjunto con Gym se podía crear un agente de RL capaz de resolver el problema con bastante sencillez. Esto mostraba la accesibilidad de estas técnicas, y con los conocimientos adquiridos pasábamos a analizar el problema principal.

Finalmente, en el capítulo 4 nos centramos en el enrutamiento de conexiones en una red real, la NSFNet. Nuestros experimentos estaban inspirados en Suárez-Varela *et al.* [1], y los diferenciamos fundamentalmente en que probábamos un escenario en el que los agentes de RL entrenaban y actuaban de forma continua. También planteamos una forma diferente de dar la información sobre la red y las peticiones que reciben los agentes, otorgándoles información del estado actual de la red (en lugar de sus estados futuros).

Acabamos por llegar a la conclusión de que el aprendizaje por refuerzo era una buena alternativa a algoritmos más tradicionales, pero es necesario profundizar en los estudios realizados. Su potencial es obvio, ya que ha obtenido resultados aceptables mientras que la configuración de estos agentes era básica y sin afinamiento destacable de sus hiperparámetros, usando un entrenamiento simple. Cabe esperar que con un estudio posterior se podría lograr mejorar su rendimiento de forma considerable, siempre y cuando se encontrara un entrenamiento que pudiera soportar la escala de las redes analizadas (problema principal observado).

Aun así, demostrábamos cómo aplicar RL profundo tenía sus ventajas con respecto a algoritmos tradicionales, ya que con solo elegir una ruta sus resultados eran, tal como mencionamos antes, mejores que si siempre se elige la misma ruta predeterminada como hace el algoritmo de la ruta más corta. Además, su flexibilidad permitía el uso de control de

admisión implementado de distintas formas sin mayor complejidad. En concreto, discriminar peticiones dependiendo de una prioridad preasignada a estas era sencillo ya que los algoritmos de RL aprendían solos las fracciones de estas que debían ser descartadas en función del estado de la red (fundamentalmente la carga de la red).

Una última observación interesante que obtuvimos en el capítulo 4 fue que el algoritmo TRPO era globalmente inferior a varios algoritmos basados en el aprendizaje de la función Q (DDQN, SARSA y PAL). No se puede asegurar con total certeza si esto siguiera siendo así de afinar minuciosamente sus hiperparámetros, pero los resultados eran claros. Lo relevante de esto es que el algoritmo TRPO es un método de gradiente de la política (*policy gradient*), no de aprendizaje de la función de valor de la acción (Q), lo que parece indicar que este último es mejor en un problema de enrutamiento dinámico. Esto es contrario a lo observado en Suárez-Varela *et al*. [1], donde el algoritmo TRPO obtenía los mejores resultados (si bien su problema era incremental y no dinámico).

5.2. Líneas futuras

A lo largo de la realización de la memoria en varios puntos de nuestro estudio nos hemos encontrado con detalles que era necesario obviar por limitaciones temporales y computacionales. En este apartado mencionamos los aspectos más relevantes que serían interesantes de analizar en profundidad en futuros estudios.

En el capítulo 3 encontrábamos una sería limitación de los algoritmos de aprendizaje por refuerzo profundo que emplean redes neuronales. Nos referimos al espacio de acción que se debe emplear. En este caso, debido a que la estructura de la red neuronal es fija, su salida no cambia. Esto dificulta mucho su aplicación en casos en los que los estados del entorno presentan un número variable de acción. Vimos que esto no era un problema con un enfoque tabular, pero al estar usando aprendizaje profundo, debíamos solventar esto de alguna manera. La opción elegida fue dejar al algoritmo aprender que acciones son nulas en cada estado. Esto es ineficiente e idealmente se debería evitar. Vimos en [22] que esto es un tópico actual de discusión sin un mecanismo bien definido para abordarlo. Aunque solventar esto de forma generalizada sería un trabajo muy considerable, una posible línea futura consiste en probar algunas de las alternativas adicionales propuestas en dicha referencia.

En el capítulo 4 es donde tuvimos que realizar la mayoría de las simplificaciones. Como planteábamos un problema con una complejidad significativa, probar solamente cuatro algoritmos de aprendizaje por refuerzo con configuración básica es muy posiblemente insuficiente. Una mayor variedad de algoritmos y configuraciones sería algo interesante de probar, además de mejorar la red neuronal empleada. También el análisis de otras redes con topologías considerablemente distintas a la NSFNet es algo a tener en cuenta. Además, una mayor variedad de escenarios podría ofrecer unos datos más detallados de lo bueno que es el aprendizaje por refuerzo en el ámbito. Aun así, sobre todo habría que centrarse en encontrar un entrenamiento más adecuado (que incluya todas las cargas de la red posibles), y podría empezarse por el propuesto a continuación.

Nuestra propuesta está inspirada en una prueba rápida realizada que simplifica las que ya mostramos en anteriores apartados. Si se considera que la fuente de tráfico es un solo nodo que genera peticiones y quiere conectarse con todos los demás (un nodo al resto, en lugar de todos con todos), se obtiene resultados muy buenos para el escenario del experimento 4.4.1. En esta situación, la probabilidad de bloqueo del RL mejora a la del algoritmo de las K rutas más cortas. Consideremos pues entrenar a los agentes en condiciones diferentes a las que se encontrarán durante su puesta a prueba. La idea es realizar un entrenamiento por fases en que los nodos que generan tráfico en la red se vayan turnando al hacerlo. Durante

un cierto tiempo, un nodo genera tráfico a los demás, y después lo hace otro. De esta forma, los agentes aprenden a enrutar minuciosamente el tráfico proveniente desde cada nodo. En la práctica esto funciona bien, pero a medida que se añaden nodos al entrenamiento empeoran los resultados. Por ello, consideramos que un posible estudio futuro debería probar esta posibilidad a fondo. La configuración de los algoritmos aquí es fundamental (especialmente el tamaño del buffer de *experience replay*, descartando el uso del TRPO).

Todo lo anterior es, sin duda, un estudio más profundo, pero tampoco ofrece aspectos nuevos a tener en cuenta. Mientras tanto, un parámetro de la red que obviamos de forma generalizada en toda la memoria es el retardo. Si bien en el capítulo 3 los enlaces tenían un coste asociado, que podría tener en cuenta el retardo, en el capítulo 4 solamente consideramos el ancho de banda de los enlaces. No es novedoso que el retardo en las comunicaciones es algo de extrema importancia, e ignorarlo es una simplificación considerable. Es por ello por lo que el fundamental punto a tener en cuenta en futuras ampliaciones de este estudio es este parámetro de las redes, y consideramos que debería ser añadido ya que los algoritmos de aprendizaje por refuerzo deberían ser capaces de tenerlo en cuenta para tomar sus decisiones (lo que podría suponer una ventaja considerable).

A modo de resumen, finalizamos este apartado enumerando las líneas futuras mencionadas en una lista:

- Buscar mejores alternativas al problema de un espacio de acción variable al usar aproximación de funciones.
- Mejorar la red neuronal empleada para implementar los algoritmos de RL para que este mejor adaptada al problema del enrutamiento de conexiones.
- Probar algoritmos de RL distintos a los ya presentados (especialmente los que no se basan en *Q-learning* y descenso del gradiente) y configurarlos de forma óptima.
- Encontrar un entrenamiento adecuado y escalable a redes de un tamaño considerable.
- Incluir el retado y estudiar su impacto en el problema del enrutamiento de conexiones.

Bibliografía

[1] J. Suárez-Varela, A. Mestres, J. Yu, L. Kuang, H. Feng, A. Cabellos-Aparicio y P. Barlet-Ros, «Routing in optical transport networks with deep reinforcement learning,» *IEEE/OSA Journal of Optical Communications and Networking,* vol. 11, nº 11, pp. 547-558, 2019.

[2] R. S. Sutton y A. Barto, Reinforcement Learning: An Introduction (Second Edition), MA: MIT Press, 2018.

[3] Y. Fujita, T. Kataoka, P. Nagarajan y T. Ishikawa, «ChainerRL,» [En línea]. Disponible en: https://github.com/chainer/chainerrl. [Último acceso: Agosto 2020].

[4] M. Plappert, «KerasRL,» GitHub, [En línea]. Disponible en: https://github.com/keras-rl/keras-rl. [Último acceso: Enero 2020].

[5] A. Kuhnle, M. Schaarschmidt y K. Fricke, «Tensorforce,» [En línea]. Disponible en: https://github.com/tensorforce/tensorforce. [Último acceso: Enero 2020].

[6] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli y J. Peters, «MushroomRL,» [En línea]. Disponible en: https://github.com/MushroomRL/mushroom-rl. [Último acceso: Enero 2020].

[7] A. Brain, «Pyqlearning,» [En línea]. Disponible en: https://code.accel-brain.com/Reinforcement-Learning/. [Último acceso: Enero 2020].

[8] I. Caspi, G. Leibovich, G. Novik y S. Endrawis, «Coach,» [En línea]. Disponible en: https://github.com/NervanaSystems/coach. [Último acceso: Enero 2020].

[9] M. Murray, «NAME RL,» [En línea]. Disponible en: https://github.com/M-J-Murray/MAMEToolkit. [Último acceso: Enero 2020].

[10] OpenAI, «Gym,» [En línea]. Disponible en: https://github.com/openai/gym. [Último acceso: Febrero 2020].

[11] V. Mnih, K. Kavukcuoglu y D. Silver, «Human-level control through deep reinforcement learning,» *Nature,* vol. 518, pp. 529-533, 2015.

[12] L.-J. Lin, «Self-improving reactive agents based on reinforcement learning, planning and teaching,» *Machine Learning,* vol. 8, pp. 293-321, 1992.

[13] H. v. Hasselt, A. Guez y D. Silver, «Deep Reinforcement Learning with Double Q-learning,» 2015. [En línea]. Disponible en: https://arxiv.org/abs/1509.06461.

[14] M. G. Bellemare, G. Ostrovski, A. Guez, P. S. Thomas y R. Munos, «Increasing the Action Gap: New Operators for Reinforcement Learning,» 2015. [En línea]. Disponible en: https://arxiv.org/abs/1512.04860.

[15] J. Schulman, S. Levine, P. Moritz, M. I. Jordan y P. Abbeel, «Trust Region Policy Optimization,» de *Proceedings of the 31st International Conference on Machine Learning (ICML)*, Lille, 2015.

[16] J. Schulman, P. Moritz, S. Levine, M. I. Jordan y P. Abbeel, «High-Dimensional Continuous Control Using Generalized Advantage Estimation,» 2018. [En línea]. Disponible en: https://arxiv.org/abs/1506.02438.

[17] E. W. Dijkstra, «A note on two problems in connexion with graphs,» *Numerische mathematik,* vol. 1, nº 1, pp. 269-271, 1959.

[18] J. A. Boyan y M. L. Littman, «Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach,» de *Advances in Neural Information Processing Systems 6 (NIPS)*, San Francisco, 1993.

[19] W. Baldwin, «Dijkstar,» [En línea]. Disponible en: https://pypi.org/project/Dijkstar/. [Último acceso: Febrero 2020].

[20] L. Shi, «Reinforcement Learning in Path Finding,» [En línea]. Disponible en: https://github.com/shiluyuan/Reinforcement-Learning-in-Path-Finding. [Último acceso: Febrero 2020].

[21] S. Raschka y V. Mirjalili, Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow, 2nd Edition, Packt Publishing, 2017.

[22] «Inconsistent action space in Reinforcement Learning,» Stack Exchange (Artificial Intelligence), [En línea]. Disponible en: https://ai.stackexchange.com/questions/9491/inconsistent-action-space-in-reinforcement-learning. [Último acceso: Julio 2020].

[23] J. Y. Yen, «Finding the K Shortest Loopless Paths in a Network,» *Management Science,* vol. 17, nº 11, pp. 661-786, 1971.

[24] A. Hagberg, D. Schult y P. Swart, «Networkx documentation,» [En línea]. Disponible en: https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.simple_paths.shortest_simple_paths.html. [Último acceso: Mayo 2020].

[25] ITU, «Interface for the optical transport network,» 2016. [En línea]. Disponible en: https://www.itu.int/rec/T-REC-G.709/. [Último acceso: Julio 2020].

[26] X. Chen, B. Li, R. Proietti, H. Lu, Z. Zhu y S. Yoo, «DeepRMSA: A Deep Reinforcement Learning Framework for Routing, Modulation and Spectrum Assignment in Elastic Optical Networks,» 2019.