UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Estudio de estrategias de Aprendizaje por Refuerzo para control de elasticidad de servicios de *streaming*

Informe de Proyecto de Grado presentado por

Micaela Larraura, Martín Rotti, Sofía Tito Virgilio

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Javier Baliosian
Matías Richart

Montevideo, 7 de diciembre de 2023

Agradecimientos

A mi madre, mi pareja y mis amigos que ya deben estar cansados de escucharme hablar de este proyecto. Y mis gatos.

A Sofi y Martín, por hacer estos últimos dos años soportables.

Micaela Larraura

A mis padres y mi hermano, por ser mi sostén en todo momento y alentarme en cada decisión tomada.

A mi novia Bruna, por el cariño de todos los días, por escucharme, aconsejarme, y acompañarme en los momentos más difíciles.

A mis abuelos, por su bondad, afecto inmenso y prepararme las mejores comidas.

A mis compañeras Micaela y Sofía por hacer que este camino sea mucho más llevadero, y por compartir alegrías y frustraciones.

A mis amigos, por siempre estar pendientes de mis avances en la carrera y por ser un pilar fundamental en mi vida.

A mis primos, tíos, suegros y el resto de mi familia por su apoyo incondicional y alegrarse con cada logro.

Por último, a la Universidad de la República, por brindarme la posibilidad de formarme, y permitirme conocer grandes personas.

Martín Rotti

A mi madre, por su amor incondicional, su apoyo y su sacrificio, por ser mi mayor confidente, por confiar en mí cuando ni siquiera yo misma era capaz de hacerlo, y porque su bondad, altruismo y resiliencia me hicieron ser quien soy.

A mis hermanos, mi cuñada, mi sobrina, Kala y el resto de mi familia, por acompañarme, por entender mis ausencias y por siempre motivarme a ser mi mejor versión.

A Emiliano Bértola y Matías Viva, por permanecer siendo certeza entre tanta incertidumbre.

A Matías Cikurel, por haber compartido codo a codo una de las etapas más desafiantes de nuestras vidas, haciendo mucho más ameno el recorrido y llenándolo de risas.

A Guillermo Dufort Y Álvarez, por regalarme sus palabras de aliento y sus abrazos cuando más lo necesité.

A Micaela Larraura, por ser una de las amistades más lindas que me dejó la facultad, y también a Martín Rotti, por soportar juntos todas mis manías y anécdotas, y por acompañarme en esta montaña rusa de emociones.

A Franco Seveso, por su amistad, que supo ser mi cable a tierra y traerme calma en medio de la tormenta, y por darme el impulso que necesito siempre que el miedo está por ganarme la carrera.

Y a la Udelar, por brindarme las herramientas necesarias para estudiar esta carrera y la oportunidad de acceder a la educación que todos merecemos.

Sofía Tito Virgilio

Por último, agradecer a nuestros tutores, Javier Baliosian y Matías Richart, por guiarnos en esta travesía, por todo su apoyo, paciencia y compromiso.

Resumen

Con el gran crecimiento de servicios que se brindan en la nube, surge el desafío de asignar recursos de manera tal de obtener un buen rendimiento en momentos de mayor demanda, sin desperdiciarlos en momentos de baja demanda. En ese sentido, surge el concepto de elasticidad, que es la capacidad de un sistema de asignar recursos de manera dinámica en base a sus necesidades.

Podemos distinguir dos tipos de elasticidad, la elasticidad horizontal, que corresponde a variar la cantidad de instancias de recursos de cómputo disponibles en el sistema, y la vertical que consiste en aumentar o disminuir características de los recursos informáticos, como el tiempo de CPU, la memoria o el ancho de banda de la red.

En este trabajo se propone atacar el problema de controlar la elasticidad horizontal de un sistema de *streaming* (reproducción de video en línea) particular. En particular, se propone lograr variar la cantidad de servidores disponibles de forma tal que se ajusten a la cantidad de usuarios que solicitan la reproducción de un video y a las características del video solicitado que puedan influir en el tiempo de servicio.

Se aborda el problema utilizando Aprendizaje por Refuerzo, una de las técnicas del área del Aprendizaje Automático que suele utilizarse para tareas de control de sistemas.

Para esto, se implementó un ambiente de desarrollo que permite entrenar y evaluar el desempeño de distintas técnicas de Aprendizaje por Refuerzo a la hora de resolver el problema abordado. Más precisamente, que permita aprender y evaluar una política que determine cuándo quitar o agregar servidores del sistema de acuerdo con su estado actual, minimizando el desperdicio de recursos y manteniendo un cierto nivel de desempeño definido sobre el tiempo de respuesta del sistema.

Se seleccionaron tres técnicas de Aprendizaje por Refuerzo a implementar: *Deep Q-Network*, *SARSA* y *Policy Gradient*, comparando su rendimiento con el de una heurística determinista. De acuerdo con los resultados observados, se concluye que dado un sistema de *streaming* con las características determinadas para el proyecto, los modelos basados en los métodos de *Policy Gradient* y *SARSA* no

dieron resultados satisfactorios habiendo sido entrenados por 1000 episodios. Sin embargo, el modelo entrenado utilizando el método *Deep Q-Network* sobre un espacio discreto de estados, en tan solo 1000 episodios logró un rendimiento superior al de una heurística basada en umbrales utilizada como línea base, siendo capaz de asignar recursos de manera eficiente, manteniendo en un rango adecuado los tiempos de respuesta hacia los clientes, y minimizando el desperdicio de recursos.

Palabras clave: Aprendizaje por Refuerzo, Control de elasticidad, *streaming*

Índice general

Capítulo 1

Introducción

Con el gran crecimiento de servicios que se brindan en la nube, surge el desafío de asignar recursos de manera tal que no se desperdicien en momentos de baja demanda, pero a la vez se provea la cantidad necesaria para obtener un buen rendimiento en momentos de mayor demanda. Para lograr esto los servicios de la nube utilizan el concepto de elasticidad, que es la capacidad de un sistema de asignar recursos de manera dinámica en base a sus necesidades.

Junto con este desafío, surge la necesidad de implementar controladores de elasticidad, encargados de gestionar los recursos de un sistema. Este control de elasticidad puede realizarse tanto de forma horizontal como vertical. El control horizontal corresponde a variar la cantidad de instancias de recursos de cómputo disponibles en el sistema, y el control vertical consiste en aumentar o disminuir características de los recursos informáticos disponibles, como el tiempo de CPU, la memoria o el ancho de banda de la red.

Un servicio particular al que se le puede aplicar control de elasticidad es el de un sistema de *streaming*, es decir, de reproducción de video en línea, que consiste de una arquitectura cliente-servidor, en la que un cliente solicita algún tipo de contenido multimedia y un servidor se encarga de proveer ese contenido de manera tal que el cliente pueda consumirlo a medida que este es descargado. En este caso el control de elasticidad tiene como objetivo lograr que los recursos disponibles se ajusten a la demanda que posea el servicio en cada momento, en este caso, a la cantidad de usuarios que solicitan la reproducción de un vídeo y a las características del video solicitado que puedan influir en el tiempo de servicio, como por ejemplo su duración o su tamaño.

En este tipo de sistemas se puede realizar tanto un control de elasticidad vertical, variando los recursos de cada servidor del sistema (velocidad de fetch, cantidad de hilos HTTP, etc.), como un control de elasticidad horizontal, variando la cantidad de servidores disponibles en el sistema.

Un tipo de sistema de *streaming* es el sistema de *streaming* de video adaptativo basado en HTTP (HAS), el cual tiene la particularidad de que cada cliente adapta la calidad de cada segmento de video que solicita en base a su percepción del estado actual de la red y del sistema de *streaming*.

El objetivo de este proyecto es trabajar sobre control horizontal de elasticidad de un sistema HAS.

En la Figura 1.1 se muestra un esquema de la arquitectura de un sistema de control de elasticidad.

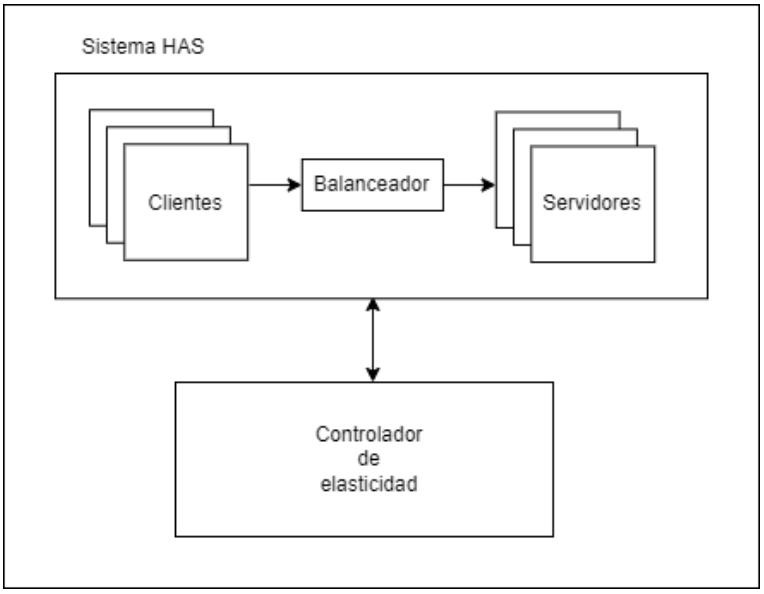

Figura 1.1: Arquitectura de un entorno de control de elasticidad de un sistema HAS.

En este proyecto se propone atacar este problema utilizando una técnica de Aprendizaje Automático llamada Aprendizaje por Refuerzo, que se basa en la interacción de un agente con un entorno de aprendizaje. En cada interacción, el agente observa el estado del entorno antes y después de ejecutar una acción sobre el mismo. En base al cambio en el estado del entorno, recibe una recompensa que indica qué tan acertada fue su decisión con respecto al objetivo de aprendizaje, y modifica su comportamiento iterativamente, buscando maximizar la recompensa total. Como resultado del aprendizaje se obtiene una política, la cual es una función que indica una acción a partir de un estado del entorno. De esta forma, la política aprendida puede ser utilizada para controlar la elasticidad del entorno.

Sin embargo, trabajar directamente sobre un sistema real conlleva distintas dificultades, tales como el costo de construir y mantener dicho sistema, y el tiempo que requeriría el entrenamiento de un algoritmo de Aprendizaje por

Refuerzo que interactúa constantemente con el sistema. Por esto, es necesario contar con un modelo de simulación del sistema real de manera tal que permita entrenar y evaluar el algoritmo en un entorno de menor costo y complejidad, y a la vez disminuya el tiempo requerido para el entrenamiento.

En este contexto, el objetivo de este trabajo es construir un ambiente de desarrollo que permita entrenar y evaluar el desempeño de distintas técnicas de Aprendizaje por Refuerzo a la hora de resolver el problema de realizar control de elasticidad horizontal de un sistema HAS. Es decir, que permita aprender y evaluar una política que determine cuándo quitar y agregar servidores del sistema de acuerdo a su estado actual, minimizando el desperdicio y manteniendo un cierto nivel de desempeño definido sobre el tiempo de respuesta del sistema.

Se seleccionaron tres técnicas de Aprendizaje por Refuerzo a implementar: *Deep Q-Network*, *SARSA* y *Policy Gradient*, con el objetivo de evaluar su desempeño a la hora de resolver el problema, comparando su rendimiento con el de una heurística determinista.

El ambiente de desarrollo antes mencionado, debe contar con un simulador de un sistema HAS y un controlador de elasticidad que interactúa con el simulador y que es utilizado tanto para el entrenamiento como para la evaluación de las políticas generadas utilizando técnicas de Aprendizaje por Refuerzo. Por ende, debe contar también con módulos de entrenamiento que interactúan con el controlador de elasticidad y el simulador para generar las políticas antes mencionadas, y con un módulo de evaluación que permita evaluar el desempeño de cada técnica para la resolución del problema estudiado.

Dado que no se encontró un simulador de un sistema HAS existente que cumpliera con los requerimientos necesarios para hacer un control de elasticidad, en un trabajo previo en el marco de una unidad curricular de la Facultad de Ingeniería de la UdelaR (Larraura, Rotti, y Tito Virgilio, 2022), desarrollamos un simulador inicial de un sistema HAS, basándonos en modelos ya existentes. Este simulador fue tomado como base para el desarrollo del simulador que finalmente se utilizó en este trabajo. Algunos componentes fueron reutilizados, mientras que otros fueron re-implementados y adaptados a las necesidades del proyecto. El resto del ambiente de desarrollo se implementó enteramente en el marco de este proyecto.

Además de construir un ambiente de desarrollo con las características antes mencionadas, se propone obtener resultados que permitan determinar si las técnicas implementadas son adecuadas o no para atacar el problema en estudio. En ese sentido, si bien no es claro si los métodos *SARSA* y *Policy Gradient* contarán con las capacidades necesarias para atacar el problema, se espera que *Deep Q-Network* posea la robustez necesaria para superar el desempeño de una política determinista, debido a la popularidad y complejidad del método.

En resumen, el objetivo general del proyecto consiste en estudiar diferentes técnicas de Aprendizaje por Refuerzo para el control de elasticidad horizontal

en servicios de *streaming*. Para ello se definen además los siguientes objetivos intermedios:

- Construir un modelo de simulación de un sistema HAS que permita entrenar y evaluar el algoritmo en un entorno de menor costo y complejidad, disminuyendo el tiempo requerido para el entrenamiento.
- Construir un ambiente de desarrollo que permita entrenar y evaluar el desempeño de distintas técnicas de Aprendizaje por Refuerzo a la hora de resolver el problema abordado. El cual debe contener e integrar el modelo de simulación, un controlador de elasticidad, y módulos de entrenamiento y evaluación.
- Utilizar dicho ambiente para implementar, entrenar y evaluar el desempeño de las técnicas: Deep Q-Network, SARSA y Policy Gradient a la hora de resolver el problema, comparando su rendimiento con el de una heurística determinista basada en umbrales.

El resto del documento se estructura de la siguiente manera. En el Capítulo 2 se presentan los antecedentes del problema abordado, y de los métodos a ser utilizados para evaluar su desempeño a la hora de resolverlo. Para ello se presenta el concepto de *streaming* adaptativo, describiendo en particular el funcionamiento de un sistema de *streaming* adaptativo basado en HTTP (HAS), se introduce el concepto de elasticidad y su aplicación a sistemas de *streaming*, se presentan el diseño y la implementación del prototipo de simulación de un sistema HAS que fue tomado como base para el desarrollo del simulador utilizado durante el proyecto, se presentan las bases del Aprendizaje por Refuerzo así como la teoría detrás de los tres métodos estudiados, y finalmente se presenta un breve resumen de algunos trabajos enfocados en implementar control de elasticidad utilizando técnicas de Aprendizaje por Refuerzo. En el Capítulo 3 se presenta la arquitectura del entorno desarrollado para la implementación, entrenamiento, evaluación y aplicación de métodos de Aprendizaje por Refuerzo para el control de elasticidad de un sistema HAS simulado. En el Capítulo 4 se describe la implementación realizada para simular el sistema HAS, incluyendo aspectos configurables y las métricas que se obtienen de cada instancia de simulación. En el Capítulo 5 se describe la implementación de cada uno de los métodos de Aprendizaje por Refuerzo, explicando las decisiones tomadas y los cambios realizados. En el Capítulo 6 se evalúan los modelos comparándolos con una política basada en umbrales, se detallan los experimentos realizados y los resultados obtenidos. Finalmente, en el Capítulo 7 se presentan las conclusiones obtenidas, se hace un análisis del proceso de desarrollo del proyecto, incluyendo las dificultades enfrentadas, y se detallan posibles mejoras y trabajos a futuro que complementan el trabajo realizado.

Capítulo 2

Revisión de antecedentes

En este capítulo se presentan los conceptos necesarios para el correcto entendimiento del proyecto, así como también se repasan trabajos previos realizados en las áreas estudiadas.

En la Sección 2.1 se introduce el concepto de *streaming* adaptativo, el cual se basa en adaptar la calidad de video provista al cliente en base a una estimación o percepción del estado de la red al momento de la transmisión. Además, se presentan tres tipos de *streaming* adaptativo y se describe en detalle un sistema en particular de *streaming* adaptativo basado en *HTTP* (HAS). Luego, en la Sección 2.2 se introduce el concepto de elasticidad, que es la capacidad de un sistema de adaptarse a las variaciones en la demanda minimizando el desperdicio, y se mencionan formas de controlar la elasticidad de un sistema de *streaming*. Por otro lado, en la Sección 2.3 se presentan un prototipo y una implementación parcial de una simulación de un sistema HAS, que serán utilizados como base en el marco del proyecto.

Además, en la Sección 2.4 se introduce el concepto de Aprendizaje Automático, haciendo especial hincapié en el Aprendizaje por Refuerzo y presentando la teoría detrás de los tres métodos estudiados: *DQN*, *SARSA* y *Policy Gradient*. Finalmente, en la Sección 2.5 se mencionan trabajos previos que utilizan Aprendizaje por Refuerzo aplicado a control de elasticidad.

2.1. *Streaming* adaptativo

En esta sección se motiva e introduce el concepto de *streaming* adaptativo como método de transmisión de video y se describen brevemente tres posibles implementaciones: *transcoding*, *stream switching* y *Scalable Video Coding (SVC)*. En el Apartado 2.1.1 se profundiza sobre una aplicación de *stream switching*, un sistema denominado *Streaming* Adaptativo basado en HTTP (HAS). Además,

se repasan trabajos previos, presentando un prototipo de simulación de dicho sistema y parte de su implementación.

Se pueden identificar dos métodos de transmisión de video: transmisión tradicional y transmisión adaptativa (González, Castellanos, Guzmán, Arce, y Guerri, 2016).

La transmisión tradicional abarca aquellos métodos en los que el archivo de video es transmitido al cliente a una tasa de *bits* constante, a través de una conexión persistente entre el origen y el destino de la transmisión. Un ejemplo de este tipo de transmisión sería la descarga progresiva, en la que el contenido es descargado de manera progresiva y almacenado en un *buffer* local del cliente. Luego, cuando el *buffer* contiene datos suficientes se comienza con la reproducción, y si en algún momento la velocidad de reproducción excede la velocidad de descarga, entonces es necesario retrasar la reproducción del video hasta tener suficientes datos en el *buffer* de reproducción para poder continuar. Una desventaja de este tipo de métodos es que no tienen capacidad de reacción ante congestiones en la red, por lo que la calidad del servicio prestado depende del estado de la red en el momento de la transmisión. Es a partir de esta necesidad que surgen los métodos de transmisión adaptativa.

La transmisión adaptativa consiste principalmente en adaptar la calidad de video provista al cliente en base a una estimación o percepción del estado de la red al momento de la transmisión. En este sentido, existen tres propuestas de implementación: *transcoding*, *stream switching* y *SVC (Scalable Video Coding)*.

Transcoding consiste en traducir de una codificación a otra para alcanzar una tasa de *bits* específica. Esto tiene un alto costo y por lo tanto no es escalable a un sistema de tiempo real en el que la transcodificación tiene que realizarse para cada cliente.

Stream switching consiste en que el servidor mantenga distintas representaciones (codificaciones en distintas tasas de *bit*) del mismo video, las cuales están a su vez segmentadas, de manera tal que un algoritmo se encargue de elegir la representación del siguiente segmento a ser descargado, intentando que sea la que mejor se alinee al ancho de banda disponible en ese momento. Esta implementación permite disminuir el tiempo de procesamiento, pero deben tenerse en cuenta requerimientos de almacenamiento y transmisión, además de la limitante de contar con un conjunto discreto de representaciones.

Finalmente, *SVC* consiste en permitir la codificación de un video creando diferentes representaciones en forma de capas, que pueden ser integradas en un único flujo de *bits*. La representación más básica del video está contenida en la capa base, la cual corresponde a la representación de menor calidad. Además, se generan múltiples capas de mejora, que proveen un incremento en la calidad de video al ser añadidas a la capa base. *SVC* permite al servidor adaptar la tasa de *bit* agregando o removiendo capas del flujo de video, basado en la estimación del ancho de banda disponible. Este esquema en capas puede proporcionar una mayor robustez durante la transmisión de video a través de redes con fluctuaciones

continuas del ancho de banda disponible y altas tasas de pérdida de paquetes.

En el Apartado 2.1.1 se presenta y describe en detalle un caso particular de sistema de *streaming* adaptativo, el cual está basado en HTTP y utiliza el método de *stream switching*. Dicho sistema será el que se utilizará como base para el control de elasticidad durante el desarrollo del trabajo.

2.1.1. *Streaming* adaptativo basado en HTTP

Un sistema en particular de *streaming* adaptativo sobre internet que utiliza el método de *stream switching* es el denominado HAS, por sus siglas en inglés: *HTTP-based adaptive streaming*.

Es un sistema cliente-servidor, en el que el servidor almacena distintas representaciones de un mismo video, y en donde cada video a su vez está dividido en segmentos de igual duración. De esta manera segmentos de diferentes representaciones se pueden concatenar para obtener un video válido. Un ejemplo de esto puede verse en la Figura 2.1, donde se ilustra un video de 12 segundos codificado en 3 representaciones distintas y dividido en segmentos de 2 segundos, y una posible concatenación del video original a partir de segmentos de diferentes representaciones.

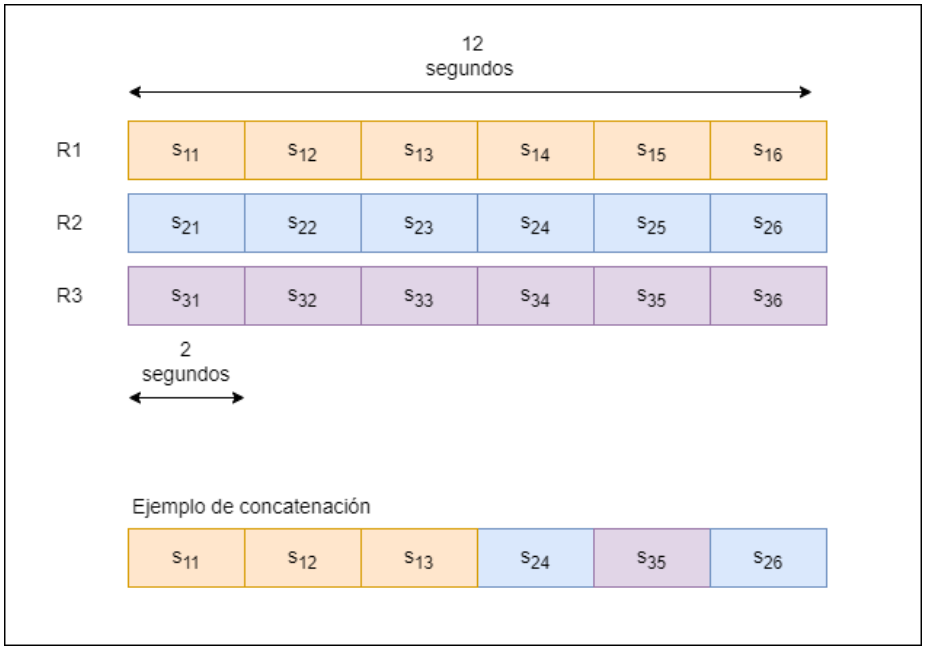

Figura 2.1: Video segmentado y codificado en 3 representaciones.

De esta forma, como se observa en la Figura 2.2, cuando un cliente se conecta al servidor, este le brinda información sobre los videos disponibles y sus respectivas representaciones. Luego, en cada interacción el cliente selecciona la representación del siguiente segmento de video a solicitar. Cada uno de los segmentos descargados es almacenado en un *buffer* de reproducción a la espera de ser reproducido. La elección de dicho segmento se realiza a partir de un algoritmo adaptativo, que ejecuta en el cliente, y en base a información de su entorno, como pueden serlo el estado del *buffer* de reproducción y su percepción del estado de la red, elige la representación del próximo segmento.

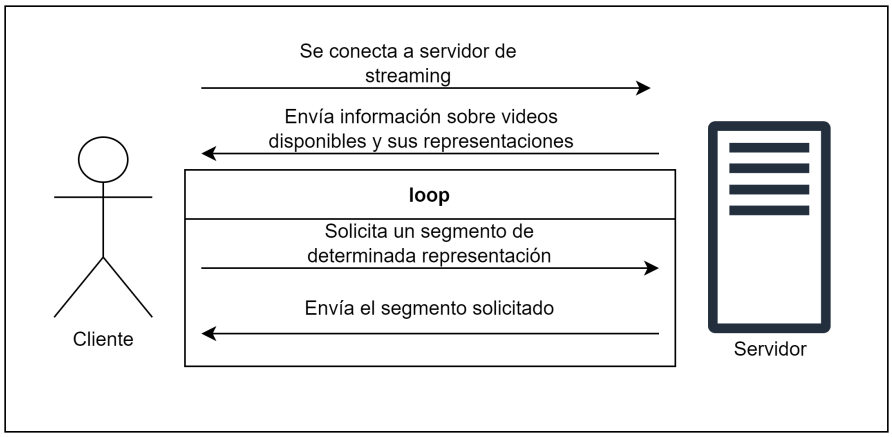

Figura 2.2: Diagrama de un sistema HAS.

Esta adaptación se hace con el objetivo de presentarle al usuario que está viendo el video la mejor experiencia posible, teniendo en cuenta la congestión del servidor y el estado de la red. Sin perder de vista que, para lograr esto, a veces es necesario sacrificar calidad de video para evitar interrupciones en la reproducción.

2.2. Elasticidad de sistemas de *streaming*

Con el gran crecimiento de servicios que se brindan en la nube, como es el caso de servicios de *streaming*, surge el desafío de asignar recursos de manera tal que no se desperdicien en momentos de baja demanda, pero a la vez se provea la cantidad necesaria para obtener un buen rendimiento en momentos de mayor demanda. A esta capacidad de los sistemas de asignar recursos de manera dinámica en base a sus necesidades, se la conoce como elasticidad.

En este contexto, al módulo encargado de gestionar los recursos del sistema se le

denomina entonces controlador de elasticidad, y dicho control puede realizarse tanto de forma horizontal como vertical. El control horizontal que corresponde a variar la cantidad de instancias de recursos de cómputo disponibles en el sistema, y el control vertical que consiste en aumentar o disminuir características de los recursos informáticos, como el tiempo de CPU, la memoria o el ancho de banda de la red.

En particular, un sistema de *streaming* es un ejemplo de sistema al que se le puede aplicar control de elasticidad, buscando ajustar los recursos disponibles a la demanda que posea el servicio en cada momento.

En este tipo de sistemas se puede realizar tanto un control de elasticidad vertical, variando los recursos de cada servidor del sistema, como por ejemplo aumentando la velocidad con la que recupera los archivos solicitados, la capacidad de concurrencia, o la velocidad de drenado hacia la red. Además, se puede realizar un control de elasticidad horizontal, variando la cantidad de servidores disponibles en el sistema.

En este caso se realizará un control de elasticidad horizontal de un sistema de *streaming* adaptativo basado en *HTTP*. Es decir, se buscará ajustar la cantidad de servidores disponibles en base a la demanda del sistema, buscando mantener un tiempo de respuesta que no perjudique la experiencia de usuario, en el sentido de que no genere interrupciones en la reproducción de los videos.

Dada la complejidad y costo de tratar con un sistema real, es necesario contar con un simulador del sistema que facilite la experimentación. Es por esto que en (Ott, Miller, y Wolisz, 2017 June), con el propósito de facilitar la experimentación y comparación de resultados asociados a este tipo de sistemas, los autores presentan el diseño y la implementación de un modelo de simulación para aplicaciones basadas en HAS, incluyendo tanto el cliente como el servidor, y permitiendo la integración de diferentes algoritmos adaptativos. Además, en un trabajo previo, introducen una máquina finita de estados que define el comportamiento de un cliente HAS. Por otra parte, describen el funcionamiento del servidor HAS como el de un servidor web que recibe y responde solicitudes de transacción de los clientes. Por lo que, en base a los resultados presentados en (Ott y cols., 2017 June) y a un modelo de servidor web presentado en (Van der Mei, Hariharan, y Reeser, 2001), en un trabajo previo realizado en el contexto de una unidad curricular de la Facultad de Ingeniería de la UdelaR (Larraura y cols., 2022), diseñamos e implementamos un prototipo de simulación para un sistema HAS, dado que no se encontró un simulador existente que cumpliera con los requerimientos necesarios para el proyecto.

En la Sección 2.3 se presenta el prototipo del sistema completo y en el Apartado 2.3.1 se describe la implementación del servidor, insumos que se utilizaron como base para el trabajo actual.

2.3. Simulación de un sistema HAS

El prototipo de simulación diseñado en (Larraura y cols., 2022) contiene dos componentes principales, que son: el modelo de simulación del cliente y el modelo de simulación del servidor.

En la Figura 2.3 se puede observar un esquema del modelo del cliente, presentado en (Larraura y cols., 2022) y basado en la máquina finita de estados presentada en (Ott y cols., 2017 June).

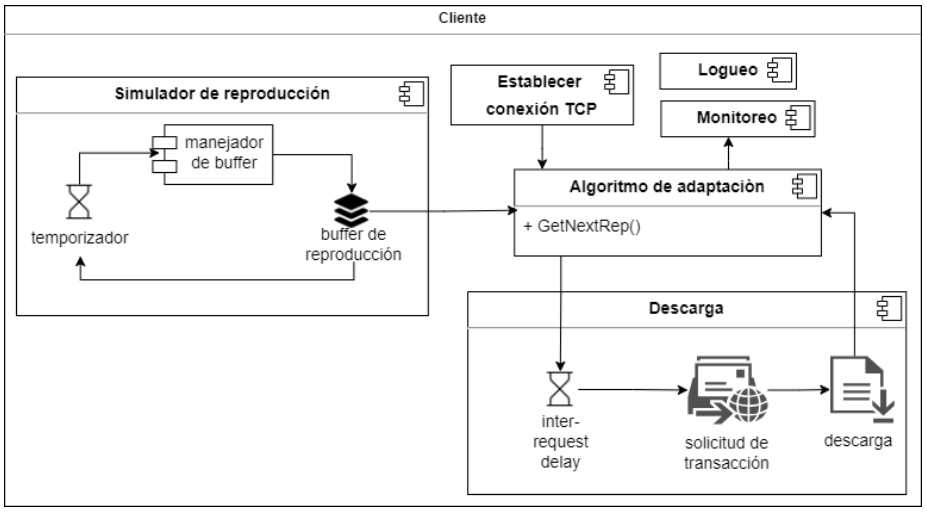

Figura 2.3: Prototipo de un cliente HAS, fuente: (Larraura y cols., 2022).

En el centro del modelo se encuentra el algoritmo adaptativo, que es invocado siempre que el cliente debe solicitar un nuevo segmento de video al servidor. Dicho algoritmo, en base a información que recauda tanto del cliente como del estado de la red, elige la representación del siguiente segmento a solicitar, y además, define si corresponde introducir un *inter-request delay*, es decir, cuánto se debe esperar antes de solicitar el próximo segmento al servidor.

Luego, se tienen los componentes encargados de simular la descarga y la reproducción del video solicitado.

Para simular la descarga de cada segmento de video, luego de invocar al algoritmo adaptativo, se inicializa un temporizador con el *inter-request delay* correspondiente, simulando la espera entre solicitudes. Cuando ese temporizador finaliza, se procede a enviar la solicitud de transacción al servidor, y se pasa a un estado de espera, en el que se espera la respuesta del servidor. Cuando se obtiene la respuesta del servidor, el segmento solicitado es depositado en el

buffer de reproducción del cliente, y se vuelve a invocar al algoritmo adaptativo para solicitar un nuevo segmento.

Luego, el componente que se encarga de simular la reproducción del video contiene un manejador de *buffer*, que se encarga de ir a buscar al *buffer* de reproducción el próximo segmento a reproducir. En caso de que no haya segmentos en el *buffer*, invoca al algoritmo adaptativo para solicitar el próximo segmento. En caso de que sí haya segmentos en el *buffer*, toma el siguiente e inicializa un temporizador igual a su duración, simulando su reproducción. Una vez que termina el temporizador, se repite el proceso.

Notar que el algoritmo adaptativo se invoca cada vez que se agrega un segmento al *buffer* de reproducción (es decir, cada vez que se descarga un segmento), y cada vez que al intentar reproducir el próximo segmento, el *buffer* de reproducción se encuentra vacío.

Finalmente, existen componentes encargados de establecer la conexión TCP con el servidor, monitorear el estado del cliente y de la red para alimentar al algoritmo adaptativo, y loguear información para su posterior procesamiento.

Por otro lado, en la Figura 2.4 se puede observar un esquema del modelo del servidor presentado en (Larraura y cols., 2022) y basado en el modelo de servidor web presentado en (Van der Mei y cols., 2001).

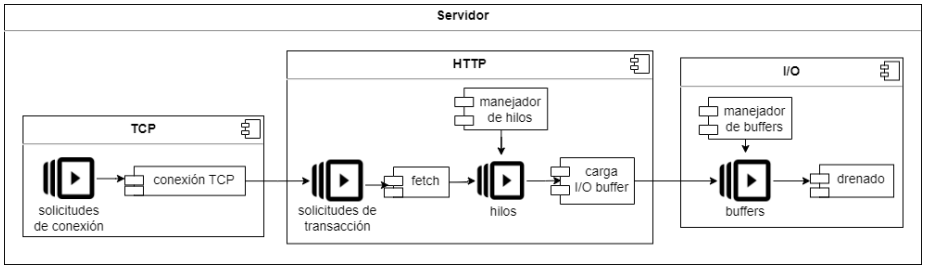

Figura 2.4: Prototipo servidor HAS, fuente: (Larraura y cols., 2022).

En este caso se pueden observar tres componentes: el componente *TCP* encargado de establecer las conexiones, el componente *HTTP* encargado de atender las solicitudes y obtener el segmento de archivo solicitado y depositarlo en un *buffer* de entrada/salida del servidor, y el componente de entrada/salida encargado de enviarlo a través de la red.

El componente *TCP* se encarga de recibir las solicitudes de establecimiento de conexión, simular el establecimiento de la conexión, y redirigir las solicitudes de transacción al componente *HTTP*. En caso de que al recibir una solicitud, el servidor ya posea la cantidad máxima de conexiones simultáneas que puede mantener, la conexión es rechazada.

El componente *HTTP* recibe las solicitudes de transacción, que son almacenadas en una cola de espera *HTTP* a la espera de ser atendidas por un hilo *HTTP*. En caso de que dicha cola esté completa, se rechaza la solicitud de transacción y se cierra la conexión con el cliente. En cuanto se libera un hilo *HTTP*, éste atiende la siguiente solicitud en espera y obtiene el segmento de archivo correspondiente. Luego de obtenido el segmento de archivo, el hilo se queda a la espera de que le sea asignado un *buffer* de entrada/salida en el que depositarlo. Se debe tener en cuenta que si el tamaño del segmento de archivo solicitado excede la capacidad del *buffer* de entrada/salida, entonces el hilo deberá depositar el segmento de archivo en el *buffer* de entrada/salida por partes y esperar a que estas sean drenadas hacia la red, hasta depositar la última parte en el *buffer*, quedando recién entonces liberado para atender otra solicitud.

Finalmente, luego de que el segmento de archivo solicitado fue depositado (total o parcialmente) en el *buffer* de entrada/salida, el componente de entrada/salida se encarga de drenar hacia la red el contenido de los *buffers*. Para esto se utiliza un recurso de drenado, que es único, y debe ser compartido por todos los *buffers* a la espera de ser drenados. La asignación del recurso de drenado se realiza siguiendo una política *round-robin* entre los *buffers* que estén esperando para utilizarlo. Cada vez que el recurso le es asignado a un *buffer*, este puede drenar una porción de su contenido equivalente a un bloque de archivo, por lo que puede ser necesario adquirir el recurso de drenado múltiples veces antes de drenar completamente el contenido del *buffer* hacia la red. Una vez que se termina de drenar completamente un *buffer*, este vuelve a quedar disponible para que un hilo *HTTP* cargue contenido en él (ya sea un nuevo hilo, o un hilo que lo esperaba para continuar cargando partes de un mismo archivo).

En la Figura 2.5 se presenta mediante un esquema un ejemplo del proceso de drenado de un segmento de archivo hacia la red. En el ejemplo, el tamaño del segmento solicitado es de seis bloques de archivo, y la capacidad del *buffer* de entrada/salida es de cuatro bloques de archivo. Por lo que el hilo *HTTP*, cuando le es asignado el *buffer* de entrada/salida, carga los primeros cuatro bloques de archivo en el *buffer* y se queda a la espera de que estos cuatro bloques sean drenados hacia la red. Tener en cuenta que para esto, el recurso de drenado debe ser asignado al *buffer* de entrada/salida cuatro veces. Una vez que los primeros cuatro bloques de archivo fueron drenados hacia la red, el hilo *HTTP* carga los últimos dos bloques de archivo en el *buffer* y queda libre para atender una nueva solicitud. Mientras tanto, el *buffer* se queda a la espera de que le sea asignado el recurso de drenado otras dos veces, para terminar de drenar el segmento de archivo solicitado y recién entonces quedar disponible para ser cargado por un nuevo hilo *HTTP*.

En el trabajo previo, además, en base al prototipo descrito anteriormente, se implementaron versiones básicas tanto del modelo del cliente como del servidor. Para ello se utilizó la herramienta de *MATLAB*, *Simulink*, que permite crear simulaciones, e incluye estructuras básicas ya implementadas que facilitan la construcción de sistemas más complejos. Sin embargo, en el trabajo previo se

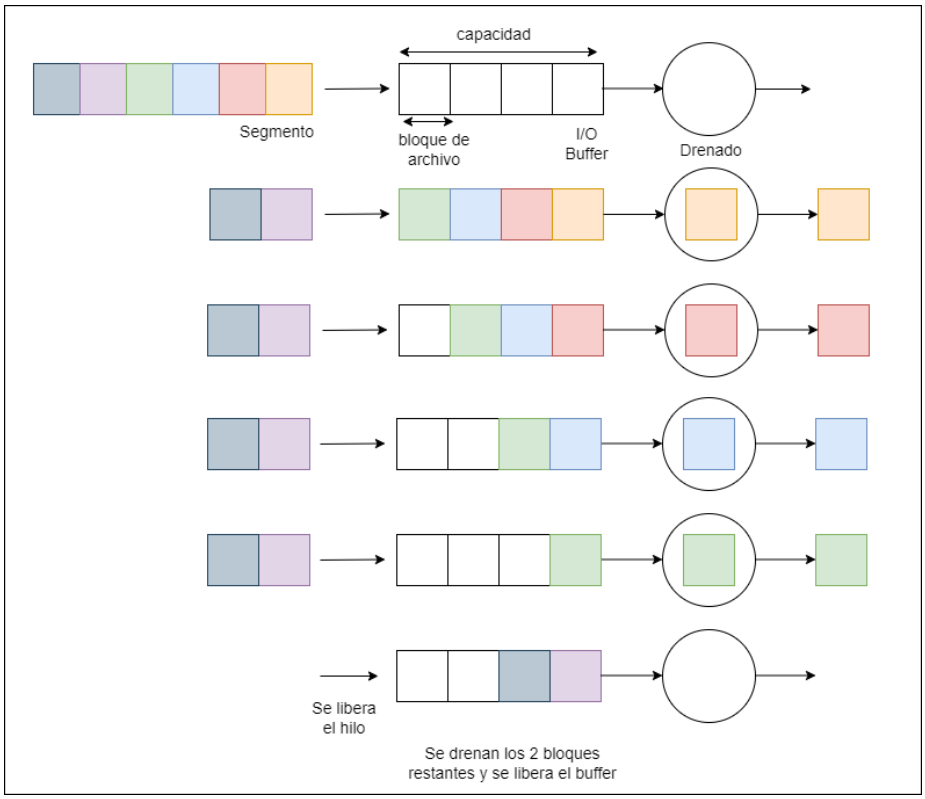

Figura 2.5: Ejemplo de drenado de un segmento de archivo.

realizó una implementación en la que cada cliente y cada servidor se representan mediante un bloque de *MATLAB*, y por lo tanto en caso de querer disponer de varios clientes y/o servidores es necesario colocar explícitamente un bloque de *MATLAB* por cada cliente y/o servidor del sistema simulado. Si bien esta implementación permite representar sin mayores dificultades los servidores de un sistema, no es escalable a la hora de representar un conjunto diverso de clientes. Es por esto que en el Apartado 2.3.1 se detalla la implementación del servidor, omitiendo la implementación del cliente, ya que esta última no fue utilizada como parte del proyecto.

2.3.1. Implementación del servidor

El simulador implementado maneja un único servidor, que fue implementado enteramente usando *Simulink*, y es consumido como una librería para que pueda ser replicado de manera sencilla a la hora de simular sistemas más complejos.

La implementación individual del servidor consiste en un bloque con una entrada y dos salidas. La entrada representa una solicitud de transacción, la primera salida la cantidad de solicitudes rechazadas, y la segunda salida representa la respuesta del servidor a las solicitudes que no fueron rechazadas. Tanto la solicitud de transacción como la respuesta del servidor son modelados con el mismo tipo de datos denominado entidad. Cada entidad tiene asociado un conjunto de datos denominados atributos, algunos fijos y otros que se irán modificando durante la simulación.

A continuación se describe la implementación de los componentes del servidor descritos en el prototipo.

La implementación del componente *TCP* se muestra en la Figura 2.6. Cuando la entidad, que representa la solicitud de transacción, ingresa al servidor, pasa primero por un bloque denominado *output switch*, que recibe entidades por un puerto de entrada y, en base a un criterio configurable, dirige a las entidades por alguna de sus salidas. En el caso del *switch* inicial, utiliza un atributo de la entidad para saber si es la primera transacción de un cliente, y por lo tanto si debería realizar la conexión *TCP*. En caso afirmativo, la entidad pasará al siguiente *switch*. Este *switch* desemboca en dos posibles caminos: el bloque de tipo servidor denominado *TCP*, que realiza la simulación del establecimiento de la conexión *TCP*, o el bloque que representa el cierre de conexión: *Close connection*. Para tomar esta decisión, se utiliza el dato que indica la capacidad del servidor para establecer conexiones *TCP* simultáneas. Si la carga actual del bloque es menor a su capacidad, el *switch* redirigirá a la entidad a *TCP*, y en caso contrario a *Close connection*, que destruye la entidad. Cada entidad destruida en esta instancia se cuenta para el total de conexiones rechazadas.

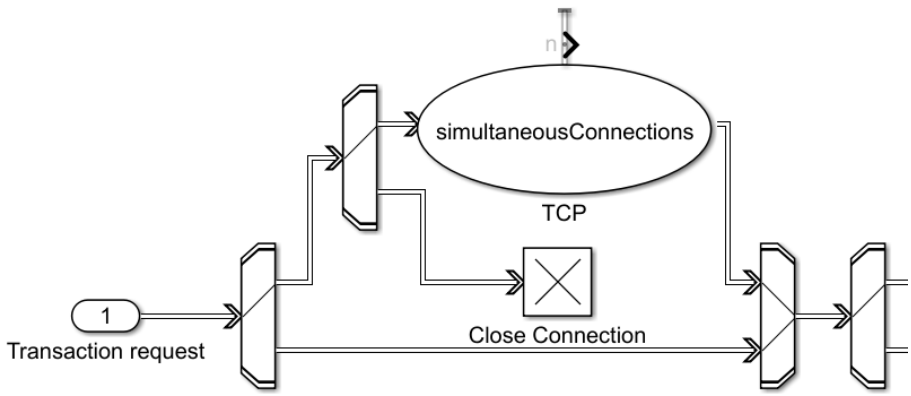

Figura 2.6: Implementación de establecimiento de conexión *TCP*, fuente: (Larraura y cols., 2022).

Haya pasado o no por la simulación del establecimiento de conexión *TCP*, mientras no se haya rechazado la conexión, la entidad pasa a la siguiente parte del simulador, que equivale al componente *HTTP* del prototipo. Esta parte se muestra en la Figura 2.7. Se realiza un proceso similar al anterior, en el que la entidad avanzará hacia la cola *HTTP Queue*, a menos que la carga de la cola sea igual a su capacidad, en cuyo caso la entidad será destruida. Esta destrucción de la entidad simula el rechazo de conexión, casos que también se suman para el total de conexiones rechazadas.

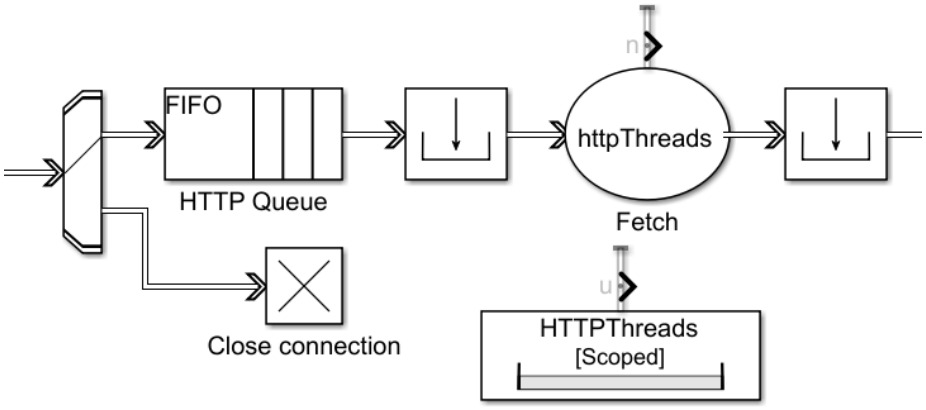

Figura 2.7: Implementación de procesamiento HTTP, fuente: (Larraura y cols., 2022).

HTTP Queue sigue una filosofía FIFO, y una vez que la entidad sea liberada pasará a obtener un recurso de lo que se llama *Resource Pool*. En particular, esta *Resouce Pool* se llama *HTTPThreads*, y representa los hilos *HTTP* disponibles. Una entidad en *HTTP Queue* soló avanzará si hay un recurso de *HTTPThreads* disponible. Una vez obtenido este recurso, se simula el *fetch* con el bloque servidor *Fetch*. El tiempo de servicio del servidor se calcula usando el tamaño del archivo o segmento a buscar (información obtenida de un atributo en la entidad), y la velocidad de *fetch* (que es una variable global configurable del simulador).

Una vez simulado el *fetch* del segmento de archivo solicitado, se pasa al componente I/O del prototipo. Esta última parte de la implementación se puede visualizar en la Figura 2.8. Para representar los *buffers* de entrada/salida, recurso que se obtiene una vez simulado el *fetch*, se usa otro *Resource Pool*, llamado *ioBuffers*. Luego de obtenido un *buffer* de entrada/salida, se pasa al bucle que representa la asignación de los *buffers* al recurso único de drenado. En primer lugar, se hacen los chequeos necesarios para verificar si lo que será depositado en

el *buffer* es la última parte de ese segmento, y en tal caso liberar el hilo *HTTP* (es decir, liberar el recurso *HTTPThreads*). Este chequeo se hace revisando un atributo de la entidad que indica lo que falta por drenar de ese segmento, y comparándolo con la capacidad del *buffer*, que es una variable global configurable.

Luego pasa a una cola FIFO que representa a los *buffers*, y cuya capacidad representa la cantidad de *buffers* disponibles. Si una entidad está en la cola que representa los *buffers*, significa que el segmento o parte del segmento está cargado en un *buffer* a la espera de obtener el recurso de drenado.

El bloque que representa el recurso de drenado tiene capacidad 1, debido a que este recurso solo puede atender un *buffer* a la vez. Cuando una entidad sale de la cola que representa los *buffers*, llega a este bloque, que tiene un tiempo de servicio configurable que representa el tiempo que tarda el servidor en drenar un bloque de archivo. Además de simular el drenado de un bloque de archivo, el servidor modifica los atributos necesarios de la entidad para reflejar si falta drenar parte del segmento, y en caso afirmativo, cuánto falta. En caso de que sea necesario seguir drenando parte del segmento, la entidad vuelve al comienzo del bucle. En caso contrario, se libera el recurso *ioBuffers*, y la entidad pasa a representar la respuesta del servidor, saliendo por la segunda salida, finalizando la simulación del procesamiento de la solicitud de transacción por parte del servidor.

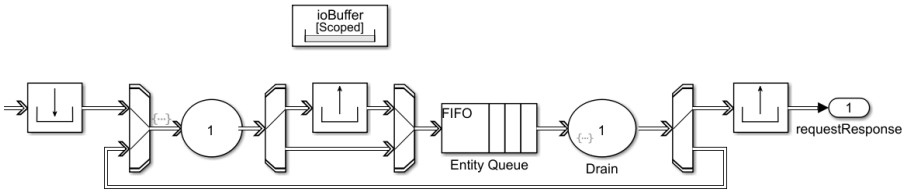

Figura 2.8: Implementación de procesamiento I/O, fuente: (Larraura y cols., 2022).

Por otro lado, notar que en esta implementación la política de asignación de *buffers* de entrada/salida al recurso de drenado no es estrictamente *round-robin*, sino que los *buffers* de entrada/salida que están esperando por el recurso de drenado se almacenan en una cola FIFO, cuando llega su turno, les es asignado el recurso de drenado para que drenen un bloque de archivo, y en caso de necesitar seguir drenando vuelven al final de la cola a esperar su turno nuevamente. Por lo que se corresponde con una versión simplificada del prototipo original del servidor.

Finalmente, notar que el comportamiento se simula en base a un conjunto de parámetros configurables que permiten determinar la cantidad y la calidad de

los recursos con los que cuenta el servidor. Parámetros que van desde la cantidad de hilos de procesamiento *HTTP* disponibles hasta las velocidades de fetch y drenado. Por lo que una de las principales ventajas de esta implementación es que modela el comportamiento de un servidor web con el nivel suficiente de detalle para permitir representar servidores de diferentes características, brindando además la posibilidad de hacer un control de elasticidad del sistema.

2.4. Aprendizaje por Refuerzo

Una forma de atacar el problema de control de elasticidad es utilizando una técnica de Aprendizaje Automático llamada Aprendizaje por Refuerzo. El Aprendizaje Automático es una rama de la Inteligencia Artificial en la que se construyen algoritmos que, basados en la experiencia y con el uso de grandes cantidades de datos, aprenden a dar una mejor respuesta. Dentro del Aprendizaje Automático existen tres tipos de modelos: Aprendizaje Supervisado, Aprendizaje No Supervisado y Aprendizaje por Refuerzo.

En el Aprendizaje Supervisado se parte de un conjunto de datos etiquetados previamente, donde cada dato consiste de una entrada junto con la salida esperada al aplicar el algoritmo. Luego, a medida que se van ingresando nuevos datos de entrada, y utilizando los datos etiquetados, se hacen ajustes al modelo para adecuarlo hasta que se considere que haya aprendido lo suficiente. De esta manera, el objetivo de este tipo de aprendizaje es generalizar las respuestas del algoritmo para que retorne una salida correcta para datos de entrada que no estén presentes en los datos etiquetados.

En el aprendizaje no supervisado, no se cuenta con ejemplos de datos etiquetados. El objetivo es poder encontrar una agrupación adecuada de acuerdo a algún patrón o estructura que posean los datos de entrada, sin ninguna intervención humana. Una de las técnicas principales de este modelo es la agrupación en clústeres, en las que se agrupan los datos con características similares en diferentes subconjuntos. Otra técnica es la de asociación, la cual consiste en crear reglas para detectar relaciones entre distintas variables de los datos, permitiendo encontrar datos que generalmente ocurren juntos.

Por otra parte, el Aprendizaje por Refuerzo se basa en la existencia de un agente que realiza acciones sobre un entorno, provocando un cambio de estado y obteniendo una recompensa que indica una valoración sobre la acción que tomó. El agente, a medida que va adquiriendo experiencia sobre el entorno, debe descubrir qué acciones producen la mayor recompensa al tomarlas. Se trata de un aprendizaje basado en ensayo y error ya que el agente comienza a actuar sin conocimiento del entorno, debiendo recorrer el espacio de estados tomando acciones sin saber si estas lo llevarán a una mejor o peor recompensa final. Este método plantea el desafío de decidir tomar una acción basándose en el conocimiento adquirido o tomar otra que no haya tomado antes pero

que lo pueda ayudar a tomar mejores decisiones en el futuro. Esto se conoce como explotación vs. exploración y es importante lograr un equilibrio entre las dos para poder obtener un buen resultado. Explotar significa que el agente, dado un estado y las posibles acciones a tomar, elige la acción que tiene mayor recompensa según lo aprendido hasta el momento, mientras que explorar se refiere a decidir tomar una acción aleatoria pudiendo no ser la que le otorgue la recompensa máxima en ese momento.

Existen varias estrategias para abordar el desafío de elegir explotar o explorar. Una de ellas puede ser definir un número pequeño fijo ϵ y cada vez que se tenga que tomar una acción, la exploración tendrá una probabilidad ϵ y la explotación una probabilidad 1 - ϵ. De esta manera se explota la mayor parte del tiempo, dejando una pequeña posibilidad de exploración (Sutton y Barto, 2018). Otra estrategia es la de comenzar explorando la mayor parte del tiempo y a medida que el agente va adquiriendo conocimiento del entorno aprovechar más la explotación, explorando cada vez menos (Bitsakos, Konstantinou, y Koziris, 2018). Esta estrategia es similar a la primera ya que la decisión se toma según un número ϵ, con la diferencia que en este caso el ϵ empezará siendo alto e irá decayendo hasta llegar a ser muy pequeño.

En la Figura 2.9 se puede ver la interacción de un agente con un entorno descrito por un conjunto de estados S. El agente puede tomar una acción de un conjunto de acciones posibles A y cada vez que toma una provoca un cambio de estado en el ambiente y se le devuelve una recompensa con un valor r que indica el valor que tiene tomar la acción en el estado actual. Esta interacción provoca una secuencia de estados s_i, acciones a_i y recompensas r_i. De esta manera, en (Mitchell, 1997) definen que la tarea del agente se basa en aprender una política $\pi : S \rightarrow A$ que maximice la suma esperada de estas recompensas y generalizan este problema como un proceso de decisión de Markov, donde en cada paso de tiempo t el agente detecta el estado actual s_t, elige una acción a_t y el ambiente le responde con una recompensa $r_t = r(s_t, a_t)$, a la vez que produce un cambio de estado pasando al siguiente $s_{t+1} = \delta(s_t, a_t)$, donde dados un estado y una acción, r retorna un valor real correspondiente a una recompensa y δ un estado $s \in S$. A medida que el agente va tomando acciones y obteniendo recompensas, se va generando un valor acumulado definido de la siguiente manera:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.1}$$

donde la secuencia de recompensas r_{t+i} se genera comenzando en el estado s_t y utilizando repetidamente la política π para seleccionar las acciones. La constante γ, denominada factor de descuento, determina el valor relativo entre las recompensas futuras y las recompensas inmediatas, cumpliéndose $0 \leq \gamma < 1$. Notar que a medida que γ esté más cerca de 1, mayor es el peso que tienen las recompensas recibidas en el futuro en relación con la recompensa inmediata, mientras que si $\gamma = 0$ solo se tendrá en cuenta la recompensa inmediata. Por lo

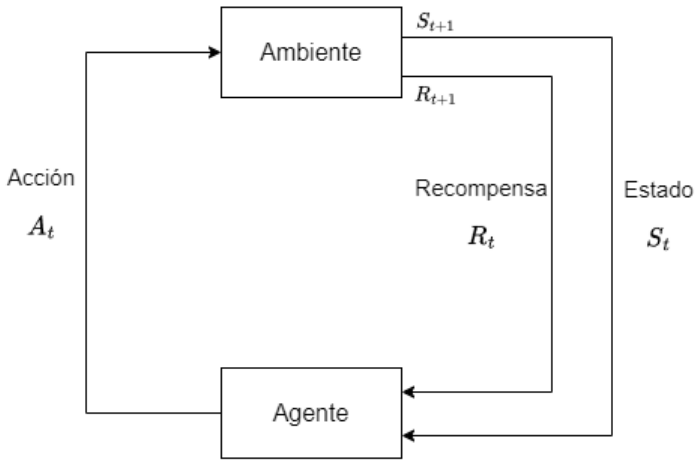

Figura 2.9: Un agente interactuando con su entorno.

tanto, la tarea del agente es aprender un política óptima denotada como π^* tal que maximice $V^\pi(s)$ para todos los estados s y se define como:

$$\pi^* = argmax_\pi V^\pi(s), \forall s \in S \tag{2.2}$$

Se denota a la función de valor V^{π^*} de la política óptima π^* como V^*.

Sin embargo, aprender la política π^* directamente, es un trabajo difícil ya que no se cuenta con datos de entrenamiento tales que dado un estado actual indiquen la acción óptima. Por otra parte, se puede aprender V^* para llegar a obtener la política óptima π^*, pero para esto es necesario que el agente conozca de antemano la función de recompensa inmediata r y la función de transición de estado γ, lo que implica predecir el resultado inmediato (recompensa y siguiente estado) para cada posible par estado-acción, algo que es imposible cuando las acciones del agente no son deterministas. Aquí se puede ver una primera clasificación entre los distintos métodos de Aprendizaje por Refuerzo: métodos basados en modelos y métodos libres de modelos. Según (Sutton y Barto, 2018), los métodos basados en modelos permiten realizar inferencias sobre cómo se comportará el entorno, pudiendo predecir la próxima recompensa a recibir y el próximo estado resultante para cada estado y sus posibles acciones a tomar. Por otra parte, los métodos libres de modelos necesitan que el agente primero lleve a cabo una acción para luego poder aprender de la experiencia.

Dentro de los métodos libres de modelos se puede hacer una distinción entre los que se basan en gradientes y los que no. Los métodos basados en gradientes parametrizan la política π con variables y toman el gradiente de la función de

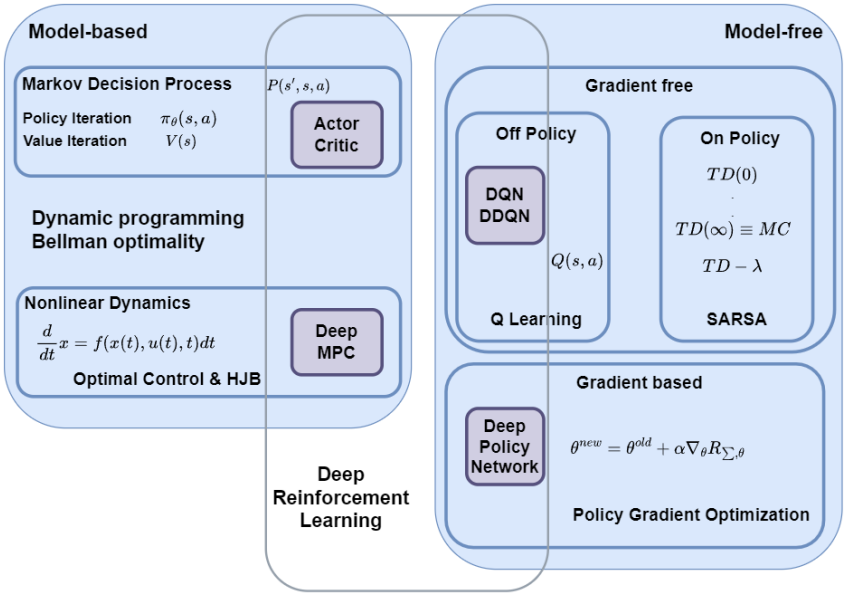

Figura 2.10: Clasificación de los métodos de Aprendizaje por Refuerzo, basada en (Brunton y Kutz, 2019).

recompensa o de la función de valor acumulado V con respecto a esos parámetros para acelerar la optimización, modelando directamente las probabilidades de las acciones, mientras que los que no se basan en gradientes no tienen la capacidad de parametrizar la política π y deben buscar la política óptima mediante la aproximación de otra función que depende del estado y la acción. Los métodos que no se basan en gradientes se clasifican en dos grupos: los métodos *Off Policy* y los *On Policy*. Si bien los dos tipos buscan aproximar iterativamente la política óptima mediante una función de calidad que depende del estado y la acción, denotada como Q(S,A), los métodos *On Policy* luego de ejecutar una acción y recibir una recompensa actualizan los valores de esta función y a la vez también la utilizan para tomar la decisión de qué acción llevar a cabo, mientras que los métodos *Off Policy* actualizan los valores de esta función pudiendo utilizar un política diferente para la toma de decisiones.

En la Figura 2.10 se puede ver la clasificación de los métodos. En las secciones 2.4.1, 2.4.2, 2.4.3 se detallan tres de ellos en profundidad: *DQN*, *SARSA* y *Policy Gradient*, ya que serán los métodos explorados más adelante para la resolución del problema.

Por último, antes de pasar a describir los distintos métodos, es importante destacar la relevancia que ha tomado en los últimos años la combinación de técnicas de Aprendizaje por Refuerzo con Aprendizaje Profundo, dando lugar a lo que se

conoce como Aprendizaje Profundo por Refuerzo. La idea básica de Aprendizaje Profundo por Refuerzo es reemplazar cualquiera de los siguientes componentes de Aprendizaje por Refuerzo: la política π, la función de valor V(s) o la función de calidad Q(s,a), por una red neuronal profunda que dado un estado y las posibles acciones retorna la mejor acción probabilísticamente que se pueda tomar con el objetivo de acelerar el proceso de aprendizaje del agente (Arulkumaran, Deisenroth, Brundage, y Bharath, 2017). De esta manera, el Aprendizaje Profundo permite que el Aprendizaje por Refuerzo se adapte a problemas de toma de decisiones que antes eran intratables, es decir, entornos con un alto número de estados y acciones.

2.4.1. *DQN*

Como se mencionó anteriormente, los métodos de Aprendizaje por Refuerzo que no se basan en modelos y no utilizan gradientes para aproximar la política óptima, definen una función que depende del estado y la acción, denotada como Q(s,a). Dentro de los métodos *Off Policy* podemos encontrar uno de los avances más importantes en Aprendizaje por Refuerzo: la técnica conocida como *Q-Learning* (Sutton y Barto, 2018). Con esta técnica, los valores de la función Q son actualizados a medida que el agente va tomando acciones de la siguiente manera:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.3}$$

A la derecha de la igualdad se puede identificar:

- $Q(s_t, a_t)$ es el valor actual en la tabla Q para el par (s_t, a_t).
- α es la tasa de aprendizaje que regula la velocidad que posee el agente para aprender. Se cumple que $0 < \alpha < 1$ y mientras más cerca esté de 1 más rápido será el proceso de aprendizaje.
- r_{t+1} es la recompensa recibida luego de realizar la acción a_t en el estado s_t.
- $max_a Q(s_{t+1}, a)$ es el valor máximo para una acción que toma Q en el siguiente estado s_{t+1}. Este valor es multiplicado por el factor de descuento γ.

En el Algoritmo 1 se presenta *Q-Learning* en su forma más simple.

Si bien esta técnica ha logrado tener éxito, su aplicabilidad se limita a entornos con baja cantidad de estados y acciones. Esto es debido a que los valores de la función Q se almacenan en una tabla y el tamaño de esta crece exponencialmente cuando el espacio se vuelve más grande, provocando que el algoritmo se vuelva más costoso computacionalmente. Por ejemplo, una realidad con 1000 estados y 500 acciones posibles para cada estado precisaría medio millón de celdas para

Algoritmo 1 Algoritmo de *Q-Learning* tomado de (Sutton y Barto, 2018)

Initialize Q(s,a), $\forall s \in S$, $a \in A(s)$, arbitrarily, and Q(terminal-state, $\cdot$) = 0
for each episode **do**
  Initialize s
  repeat
    Choose a from $A(s)$ using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a))$
    s $\leftarrow$ s'
  until s is terminal
end for

almacenar los valores de cada par estado-acción. Esto repercute no solo en la cantidad de memoria utilizada para almacenar los valores de la función sino que también el tiempo requerido para explorar cada par estado-acción se vuelve muy grande.

Como se mencionó anteriormente, un enfoque para abordar este problema es incorporar una red neuronal profunda de manera que se utilice para aproximar los valores de Q, reemplazando la tabla. De esta manera, la red neuronal puede recibir como entrada un estado y generar como salida los valores Q para todas las acciones posibles. Esta técnica es conocida como *Deep Q-Learning* y permite manejar entornos con una gran cantidad de estados y acciones. Si bien *Deep Q-Learning* es más estable que *Q-Learning*, el hecho de agregar una red neuronal profunda tiene el desafío de que la misma, al irse actualizando en cada paso, converja a la función Q óptima. En (Mnih y cols., 2015) afirman que para lograr estabilidad se presentan varios problemas: las correlaciones presentes en la secuencia de observaciones del entorno, el hecho de que pequeñas actualizaciones pueden cambiar significativamente la política y por lo tanto cambiar la distribución de los datos, y las correlaciones entre los valores de la política actual (Q) con los valores objetivos $r + \gamma max_{a'} Q(s',a')$. Para abordar estos problemas, presentan *Deep Q-Network (DQN)*, agregando dos elementos claves a *Deep Q-Learning*, en primer lugar incorporan un mecanismo denominado *experience replay* que logra aleatorizar los datos, logrando así eliminar las correlaciones en la secuencia de observaciones y suavizando los cambios en la distribución de los datos, y en segundo lugar realizan una actualización iterativa que ajusta los valores de Q hacia los valores objetivos y estos solo se actualizan periódicamente, reduciendo así las correlaciones entre los valores actuales y los objetivos.

Para llevar a cabo el *experience replay*, almacenan las experiencias del agente $e_t = (s_t, a_t, r_t, s_{t+1})$ en cada paso de tiempo t en un conjunto de datos $D_t = \{e_1, ..., e_t\}$, usualmente llamado *replay buffer*. Las actualizaciones a Q se aplican en muestras de experiencia (s, a, r, s') extraídas uniformemente al azar del conjunto de muestras almacenadas en D. La actualización en la iteración i utiliza la siguiente función de pérdida:

$$L_i(\theta_i) = (r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2 \tag{2.4}$$

donde γ es el factor de descuento que determina el horizonte del agente, θ_i son los parámetros de la red neuronal Q en la iteración i y θ_i^- son los parámetros utilizados para calcular el valor objetivo en la iteración i. Los parámetros objetivos θ_i^- solo se actualizan con los parámetros de Q, θ_i, cada C pasos, manteniéndose fijos entre las actualizaciones individuales. Es decir, que se mantendrán dos redes neuronales, la red Q con parámetros θ_i se actualizará en cada paso de tiempo t, mientras que la red $\hat{Q}$ con parámetros θ_i^- lo hará cada C pasos. En el Algoritmo 2 se presenta el flujo que permite entrenar un agente con el método *DQN*.

Algoritmo 2 Algoritmo de DQN tomado y adaptado de (Mnih y cols., 2015)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for $episode = 1, M$ **do**
  Initialize state s_1
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Set

$$y_j = \begin{cases} r_j & \text{if step j+1 is terminal} \\ r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & otherwise \end{cases}$$

    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

2.4.2. *SARSA*

El método *SARSA*, al ser también un método que no se basa en modelos y no utiliza gradientes, tiene como objetivo aprender una función de calidad Q que depende del estado y la acción, aproximando de esta manera a la política óptima. Aquí se tienen en cuenta las transiciones del par estado-acción al par estado-acción para actualizar los valores de la tabla Q luego de cada acción tomada por el agente. Este método sigue una idea idéntica a la presentada en

la introducción de este capítulo en la que la secuencia de estados, acciones y recompensas es generalizada como un proceso de decisión de Markov, pero en este caso también se tiene en cuenta el estado y la acción siguiente (Sutton y Barto, 2018). De esta manera, para actualizar los valores de la tabla Q, se tiene en cuenta la tupla de cinco elementos $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, donde r_{t+1} es la recompensa recibida por tomar la acción a_t en el estado s_t, s_{t+1} es el estado al que se llega luego de ejecutada la acción y a_{t+1} es la acción que se toma en este estado. La actualización de los valores de Q se realiza de la siguiente forma:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.5}$$

Notamos la semejanza entre esta ecuación y la ecuación de *Q-Learning* presentada en 2.3. La diferencia radica en la manera de seleccionar el valor del siguiente estado. Mientras *Q-Learning* toma el valor de una acción que maximiza el siguiente estado, *SARSA* toma el valor del par siguiente estado - siguiente acción, derivado de la política Q. En el Algoritmo 3 se presenta este método, el cual se puede ver que es muy similar al algoritmo de *Q-Learning* presentado en 1, notando que para el caso de *SARSA* se necesita realizar dos pasos de decisión de acciones para actualizar el valor de la tabla Q y dichas acciones se derivan de la misma política que se está actualizando, por este motivo se dice que *SARSA* es un método *On Policy*, mientras que en *Q-Learning* se realiza un paso de decisión de acción y luego se actualiza el valor de Q tomando el valor de la acción más alto para el siguiente estado.

Algoritmo 3 Algoritmo *SARSA* tomado de (Sutton y Barto, 2018)

Initialize Q(s,a), $\forall s \in S$, $a \in A(s)$, arbitrarily, and Q(terminal-state, $\cdot$) = 0
for each episode **do**
  Initialize s
  Choose a from $A(s)$ using policy derived from Q (e.g., ϵ-greedy)
  repeat
    Take action a, observe r, s'
    Choose a' from $A(s')$ using policy derived from Q (e.g., ϵ-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s', a') - Q(s,a))$
    s $\leftarrow$ s'
    a $\leftarrow$ a'
  until s is terminal
end for

Al igual que lo visto en *Q-Learning*, en este método se presenta el problema de que al almacenar los valores de las acciones para cada estado en una tabla, el tiempo que requiere explorar cada par estado-acción de manera que el agente adquiera un conocimiento suficiente para lograr una política óptima, se vuelve cada vez más grande a medida que se maneja una mayor cantidad de estados y acciones.

2.4.3. *Policy Gradient*

Policy Gradient, al ser un método basado en gradientes, intenta parametrizar la política π con un conjunto de variables que se irán ajustando a medida que transcurre el tiempo con el objetivo de maximizar la recompensa total recibida luego de aplicar una trayectoria de estados y acciones. Al igual que los métodos presentados anteriormente, este también puede ser representado como un proceso de decisión de Markov y se basa en realizar un ascenso (o descenso) del gradiente.

Un algoritmo de ascenso de gradiente es un proceso iterativo a través del cual se actualizan parámetros con el fin de encontrar un máximo de una función $f : \mathbb{R}^d \to \mathbb{R}$, siempre que la misma sea diferenciable. El proceso produce una secuencia de vectores $\theta_1, \theta_2, ... \theta_n$ tal que la función f es creciente a lo largo de la secuencia. En el Algoritmo 4 se presenta el proceso descrito. Se puede notar que cada vector se construye a partir del vector anterior de la secuencia, el gradiente de la función f y un parámetro α que permite controlar cuán grande es el salto entre un vector y el siguiente.

Algoritmo 4 Ascenso de gradiente tomado de (Dick, 2015)

Input: step-size $\alpha > 0$
Choose $\theta_1 = 0 \in \mathbb{R}^d$
for $time = 1, n$ **do**
    Set $\theta_{t+1} = \theta_t + \alpha \nabla f(\theta_t)$
end for

Una variante del ascenso de gradiente es el algoritmo de ascenso de gradiente estocástico, el cual se puede utilizar para maximizar la función f incluso si no es posible calcular el valor de su gradiente ∇f. Este método solo requiere que se pueda producir vectores aleatorios con un valor esperado igual a ∇f y utilizar estas estimaciones de gradiente estocástico en lugar de los gradientes reales. De esta manera, el algoritmo de ascenso de gradiente estocástico es igual al Algoritmo 4 cambiando la forma en que se crean los vectores por:

$$\theta_{t+1} = \theta_t + \alpha \nabla_t \tag{2.6}$$

donde $\mathbb{E}[\nabla_t | \theta_t] = \nabla f(\theta_t)$.

En (Dick, 2015) se afirma que el método de *Policy Gradient* es un algoritmo de aprendizaje basado en el ascenso de gradiente estocástico que tiene como objetivo aprender a maximizar la recompensa total en un proceso de decisión de Markov. Allí expresan el problema de encontrar una política óptima en términos de maximizar la función f: $\mathbb{R}^d \to \mathbb{R}$. De esta manera, parametrizando la política con un vector de variables $\theta \in \mathbb{R}^d$, la función $\pi : \mathbb{R}^d \to \prod$ retorna un política para cada vector θ. Luego, la función objetivo es J:$\prod \to \mathbb{R}, \mathbb{J}(\pi(\theta))$ a la que se le aplicará un ascenso de gradiente. Por simplicidad, se denota $J(\theta)$ a $J(\pi(\theta))$.

Para estimar el gradiente $\nabla J(\theta)$, el agente aplica la política $\pi(\theta)$ durante un cierto período de tiempo (se le llamará episodio) donde genera una secuencia de estados y acciones (s_1, a_1), (s_2, a_2), ..., (s_n, a_n). Para cualquier par estado-acción (s,a), el gradiente $\nabla_\theta \pi(s, a, \theta)$ representa la dirección en el espacio de parámetros en la que el agente movería el vector θ para aumentar la probabilidad de elegir la acción a encontrándose en el estado s. La estimación del gradiente de J es la suma de los gradientes de la política π sobre los pares estado-acción (s,a) visitados durante el episodio, cada uno multiplicado por un término que representa qué tan bien se comportó el agente después de esa acción y dividido por la probabilidad de elegir tomar la acción a en el estado s. Si $(S_t^\theta, A_t^\theta, R_t^\theta)_{t=1}^n$ representa una secuencia de estados, acciones y recompensas, obtenida de aplicar la política $\pi(\theta)$ para un episodio, además T^θ representa la primera vez que se llega a un estado terminal, $G_t^\theta = \sum_{s=t}^{T^\theta} R_s^\theta$ es la recompensa total obtenida luego del tiempo t y $\varphi_t^\theta = \frac{\nabla_\theta \pi(S_t^\theta, A_t^\theta, \theta)}{\pi(S_t^\theta, A_t^\theta, \theta)}$ se le llama vector de características compatibles en el tiempo t, entonces el siguiente vector:

$$\nabla_\theta = \sum_{t=1}^{T^\theta - 1} \varphi_t^\theta G_t^\theta \tag{2.7}$$

satisface que $\mathbb{E}[\nabla_\theta] = \nabla J_{total}(\theta)$.

Intuitivamente, el efecto de agregar esta estimación del gradiente al vector de parámetros del agente es aumentar la probabilidad de las acciones que resultaron ser buena elección y aumentar aún más la probabilidad de las acciones que funcionaron bien y rara vez se eligen. Por lo tanto, podemos usar estas estimaciones de gradiente en un método de gradiente de política simple que actualiza los parámetros de política una vez después de cada episodio. El pseudocódigo lo podemos ver en el Algoritmo 5.

Algoritmo 5 *Policy gradient* para un proceso de decisión de Markov episódico tomado de (Dick, 2015)

Input: step-size $\alpha > 0$
Choose $\theta_1 \in \mathbb{R}^d$ arbitrarily
for $t = 1, n$ **do**
    Run one episode following $\pi(\theta)$ until the terminal state is reached
    Compute ∇_{θ_t} according to equation 2.7
    Set $\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t}$
end for

Notamos la diferencia entre este método y los presentados anteriormente al momento de actualizar los valores de la política. Mientras *DQN* y *SARSA* realizan una acción estando en un estado, obtienen una recompensa y actualizan la política, *Policy Gradient* lleva a cabo una secuencia de estos pasos para luego realizar la actualización, lo que puede llevar a que el tiempo necesario para que un agente aprenda una política óptima sea más grande. Además, al utilizar ascenso por

gradiente, el método *Policy Gradient* corre el riesgo de converger rápidamente a un óptimo local en lugar de un óptimo global, lo que deriva en encontrar una política que no es la óptima. Por otra parte, los cambios en la política en *Policy gradient* (es decir, las probabilidades de las acciones) cambian más lentamente con el tiempo comparado con los basados en la función de calidad Q que debido a un cambio pequeño en los valores de acción estimados pueden cambiar drásticamente la política, lo cual es una ventaja de *Policy Gradient*.

2.5. Aprendizaje por Refuerzo aplicado a elasticidad

En esta sección se presentan algunos trabajos centrados en la aplicación de métodos de Aprendizaje por Refuerzo para el control de elasticidad de diversos servicios.

En (Bitsakos y cols., 2018), controlan la elasticidad en términos de agregar o quitar máquinas virtuales del clúster de un usuario en un sistema que contiene una base de datos bajo una carga variable y se tiene como objetivo lograr el equilibrio deseado entre el rendimiento y la latencia y a la vez mantener bajos los costos. En el trabajo presentan una solución que ataca este problema utilizando técnicas de Aprendizaje Profundo por Refuerzo, logrando lidiar con un espacio de estados complejo y de gran tamaño, adaptándose al entorno y generalizando sobre los datos de la entrada. Basándose en el algoritmo *DQN* presentado en (Mnih y cols., 2015) construyeron tres soluciones a las que le llaman: Simple *Deep Q-Learning*, *Full Deep Q-Learning* y *Double Deep Q-Learning*. Los tres algoritmos son muy parecidos entre sí, presentando pequeñas diferencias. En particular, en *Full Deep Q-Learning* se sigue los mismos pasos que en el Algoritmo 2 adaptados a este ambiente, mientras que en Simple *Deep Q-Learning* la diferencia reside en que no se utiliza la segunda red neuronal $\hat{Q}$ y en *Double Deep Q-Learning* se calcula el término y_j utilizado para realizar el descenso por gradiente de la siguiente manera:

$$y_j = \begin{cases} r_j & \text{if step j+1 is terminal} \\ r_j + \gamma \hat{Q}(s_{j+1}, argmax_a(Q(s_{j+1}, a))) & otherwise \end{cases}$$

Por otra parte, en (Tsoumakos, Konstantinou, Boumpouka, Sioutas, y Koziris, 2013) presentan un sistema que se ejecuta en tiempo real para controlar la elasticidad en la nube que integra un módulo de toma de decisiones con otro módulo de monitoreo para recopilar estadísticas que ayuden a manejar el número de máquinas virtuales de un clúster. El sistema tiene como entrada una política

definida por el usuario representada como una función de optimización y para entrenar el módulo de toma de decisiones se utiliza la técnica *Q-Learning* explicada en la Sección 2.4.1.

Además, en (Lolos, Konstantinou, Kantere, y Koziris, 2017) presentan un algoritmo que utiliza Aprendizaje por Refuerzo para atacar el mismo problema que los anteriores trabajos. El algoritmo emplea una partición adaptativa del espacio de estados del entorno según la carga de trabajo y el comportamiento del sistema para lidiar con el problema de tener una gran cantidad de estados. Para que la partición sea posible utilizan árboles de decisión, un técnica de Aprendizaje Automático supervisada, mientras que se apoyan también en *Q-Learning* para el aprendizaje del agente.

En (Bitsakos y cols., 2018) se comparan con las soluciones (Tsoumakos y cols., 2013) y (Lolos y cols., 2017) las cuales eran las mejores encontradas hasta ese momento y muestran como *Double Deep Q-Learning* las supera en dos escenarios de evaluación, tanto en el total de recompensas obtenidas por un agente como también en la capacidad de adaptarse rápidamente al entorno y predecir las siguientes acciones óptimas a tomar para mantener la elasticidad del sistema. Allí afirman que en (Lolos y cols., 2017), al utilizar árboles de decisión, se tiene un modelo más complejo que necesita un conjunto de datos más grande antes de adaptarse y encontrar la solución óptima, mientras que en (Tsoumakos y cols., 2013) si bien se logra adaptar de forma rápida, no logra encontrar la decisión óptima la mayor cantidad de veces. A la vez, en (Bitsakos y cols., 2018), al incluir redes neuronales a la solución se logra que el espacio de memoria requerido para ejecutarlo sea mucho menor que los otros enfoques ya que toda la información se almacena en los pesos de la red, al contrario de los otros que al utilizar *Q-Learning* es necesario más espacio para almacenar los valores de las acciones.

Capítulo 3

Arquitectura

En este capítulo se presenta el sistema desarrollado y se describen sus principales componentes. En la Figura 3.1 se puede observar un esquema de la arquitectura del sistema, en donde se muestran los componentes que lo integran y cómo interactúan entre ellos.

Para poder cumplir con el objetivo del proyecto, se construyó un ambiente configurable, capaz de simular un sistema HAS y controlar su elasticidad a partir del entrenamiento y la aplicación de políticas de Aprendizaje por Refuerzo.

Para ello se desarrolló un simulador de un sistema HAS que, a su vez, deberá poder usarse para entrenar algoritmos de Aprendizaje por Refuerzo. En este sentido, se deben poder realizar observaciones del estado del sistema simulado y también ejecutar acciones sobre el sistema en base a alguna política determinada, teniendo en cuenta que las posibles acciones a ejecutar en este caso serán tres: quitar un servidor, mantener la cantidad actual de servidores y agregar un servidor. Para ello, se tomó como base el modelo de simulación desarrollado y presentado en (Larraura y cols., 2022), cubriendo sus limitaciones y adaptándolo a las necesidades del proyecto.

Para ello, se implementó además un controlador de elasticidad encargado de interactuar con el simulador para realizar las observaciones y ejecutar las acciones correspondientes. En este sentido, dado que el simulador y el controlador de elasticidad están implementados en distintas tecnologías, se implementó una interfaz que permite la comunicación entre ellos.

Además, se implementó un módulo encargado del entrenamiento de las políticas siguiendo las distintas técnicas de Aprendizaje por Refuerzo estudiadas, que realiza el entrenamiento interactuando con el simulador a través del controlador de elasticidad. Finalmente, se implementó un módulo de evaluación encargado de evaluar el desempeño de la política utilizada para el control de elasticidad del sistema.

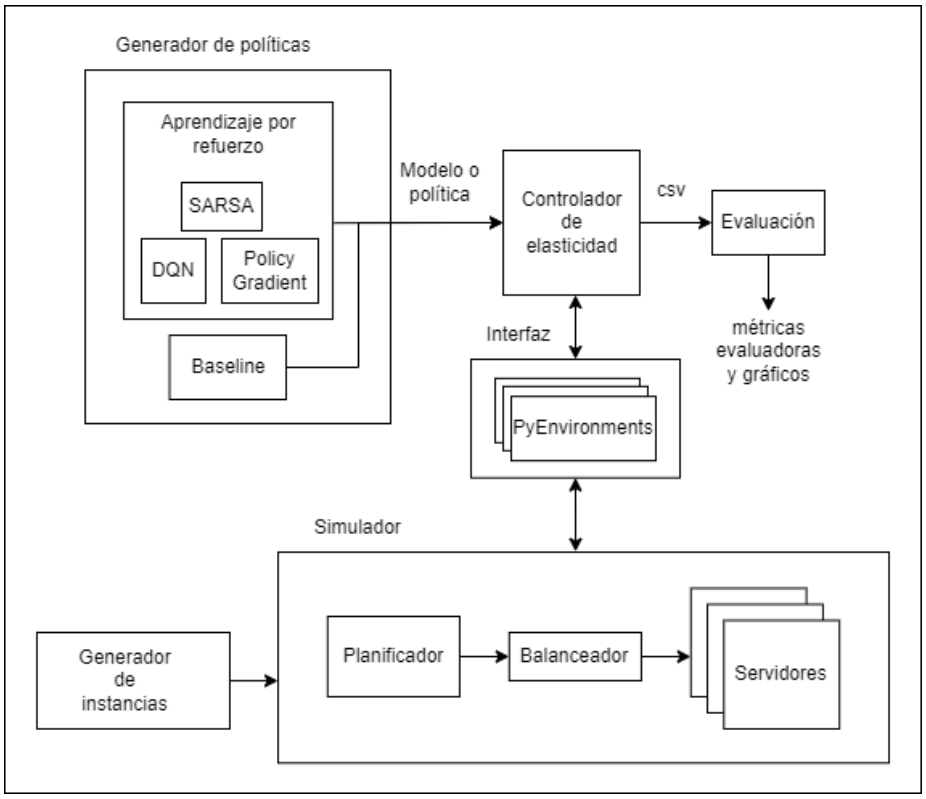

Figura 3.1: Arquitectura del sistema desarrollado.

Dentro del simulador se tienen distintos componentes que permiten simular un sistema HAS, que incluyen un planificador que simula el comportamiento de una planificación de clientes, un balanceador de carga y un conjunto de servidores. Además, el simulador es alimentado por un generador de instancias que genera archivos de configuración que definen las distintas instancias a ser simuladas.

En resumen, el componente dedicado a la generación de políticas alimenta al controlador de elasticidad, el cual interactúa con el simulador a través de la interfaz y aplica la política para controlar los recursos del sistema, generando un archivo con información que permite evaluar el rendimiento del controlador con la política aplicada sobre la instancia generada por el generador de instancias.

En las secciones 3.1, 3.2, 3.3, 3.4 y 3.5 se describen en detalle los distintos elementos y tecnologías que componen al sistema desarrollado. Además, el código fuente correspondiente se encuentra disponible en https://gitlab.fing.edu.uy/sofia.tito/proyecto-de-grado-fing-udelar

3.1. Simulador

Este componente se encarga de simular el comportamiento de un sistema HAS, descrito en el Apartado 2.1.1 de la Sección 2.1. Para ello cuenta con un planificador, que a partir de una planificación se encarga de simular el comportamiento de los clientes del sistema. Una planificación es una secuencia de clientes, dónde para cada cliente se indica el video a solicitar y en qué segundo de simulación comenzar a solicitarlo. Por otro lado, el simulador cuenta con un conjunto de servidores, donde cada servidor se encarga de simular el comportamiento de un servidor web que atiende las solicitudes de transacción realizadas por los clientes. Además, para distribuir la carga de solicitudes entre los distintos servidores disponibles en el sistema, cuenta con un balanceador que se encarga de asignar las solicitudes generadas por el planificador a los servidores, buscando equilibrar la carga de solicitudes siendo procesadas por servidor.

Este componente fue implementado utilizando una licencia estudiantil de la versión R2022a de la herramienta MATLAB. Esta herramienta, además de ser en la que estaba implementado el simulador del servidor web utilizado como base, cuenta con herramientas que facilitan la implementación del prototipo de simulación introducido en la Sección 2.3. Además, permite incluir parámetros configurables en la implementación, cuyos valores determinan el comportamiento del sistema, pudiendo ser cargados desde un archivo de configuración antes de comenzar la simulación.

Más específicamente, el servidor web, como se describe en el Apartado 2.3.1, fue implementado utilizando la herramienta Simulink, la cual cuenta con implementaciones de colas de espera, servidores, *switches*, *pools* de recursos y otros elementos que son de utilidad para simular el comportamiento del servidor, como la implementación de funciones que permiten agregar lógica personalizada a la toma de decisiones.

Por otro lado, el planificador se implementó utilizando el componente DES (*discrete-event system*) (MathWorks, 2016) de la librería SimEvents, el cual permite programar la simulación de sistemas orientados a eventos. Para lograr esto, cuenta con tres conceptos principales: *Entities*, *Storages* y eventos. Un bloque de tipo DES, tiene puertos de entrada y puertos de salida, y cada puerto tiene asociado un tipo de *Entity* que puede pasar a través de dicho puerto. Una *Entity* es un objeto que tiene asociado un conjunto de atributos que pueden ser consultados y modificados dentro del bloque y que son utilizados para la toma de decisiones. Cada *Storage* puede almacenar *Entities* de determinado tipo, y además, tiene asociado una serie de eventos que se disparan de forma automática. Por ejemplo, el evento *Entry*, que se dispara cuando llega una entidad al *Storage*, o el evento *Exit*, que se dispara cuando sale una entidad del *Storage*. A su vez, se pueden planificar otros tipos de eventos que permiten el flujo de entidades entre los distintos *Storages* y hacia fuera del bloque. Por ejemplo, se puede planificar el evento *Forward*, que reenvía una entidad de un *Storage* a

otro, u hacia fuera del bloque. También se puede planificar el evento *Generate*, que genera una entidad nueva en un *Storage* en específico. Finalmente, cada evento tiene asociada una función o manejador que se encarga de definir qué instrucciones deben ejecutarse al ocurrir dicho evento.

Esto es particularmente útil a la hora de simular el comportamiento de un cliente HAS ya que, como se vio anteriormente, sus acciones se disparan principalmente en base a eventos: al recibir un segmento, al terminar de reproducir un segmento, al encontrar vacío el *buffer* de reproducción, etc. Por lo que la herramienta utilizada permitió simular de manera bastante directa el comportamiento de una planificación de clientes.

Finalmente, los archivos de configuración que alimentan al simulador y definen los parámetros de la simulación son generados automáticamente por un generador de instancias. El mismo, a partir de la duración (en segundos) de la simulación y la cantidad esperada de nuevos clientes por minuto, se encarga de generar planificaciones aleatorias de clientes, manteniendo fijas las características de los servidores del sistema. Esto permite la generación de diversas instancias para el entrenamiento y evaluación de los distintos métodos de Aprendizaje por Refuerzo estudiados.

En el Capítulo 4 se entra en mayor detalle sobre el diseño y la implementación de todos los componentes del simulador utilizado. Para su implementación se tomó como base la implementación del modelo presentado en la Sección 2.3, cubriendo sus limitaciones y re-implementando el cliente, ya que la implementación existente no era escalable, impidiendo representar una planificación diversa y dinámica de clientes.

3.2. Generador de políticas

Este componente se encarga de implementar y entrenar distintas técnicas de Aprendizaje por Refuerzo, para lo cual se utilizó la biblioteca *Tensorflow* en su versión 2.10. En este componente se implementaron tres técnicas: *DQN*, *SARSA* y *Policy Gradient*. Además, se implementó una política con una heurística basada en umbrales para utilizar como *baseline* a la hora de evaluar los distintos métodos implementados.

Cualquiera sea la técnica de Aprendizaje por Refuerzo, se trata de entrenar un agente mediante la interacción con el entorno simulado y la aplicación del método correspondiente para actualizar la política de toma de decisiones que se está aprendiendo.

Se implementó un módulo de entrenamiento particular para cada método, cada uno es un módulo de *Python* que recibe como parámetro un archivo de configuración que define distintas características del entrenamiento. Entre ellas, la cantidad de episodios a entrenar, la duración (en segundos) de cada episodio, el

tiempo (en segundos) entre observaciones y las tasas de arribo de clientes a ser utilizadas para generar la planificación de clientes de cada episodio. Además, se definen la tasa de aprendizaje (*learning rate*) y el factor de descuento (*discount factor*) del método. Así como también otros parámetros que definen características del *replay buffer*(cuando corresponda), el *environment* a ser utilizado y también el modelo de simulador a ser utilizado como entorno de aprendizaje. Además, también se indican los nombres de los archivos de configuración del simulador para cada episodio, y si estos deben ser generados automáticamente durante el entrenamiento o si por el contrario se desea usar archivos generados previamente.

Por otro lado, cada módulo de entrenamiento genera tres archivos, un archivo de *log*, un archivo CSV con información de cada episodio, y un archivo CSV con información de cada iteración. Estos archivos son de utilidad para la depuración y la evaluación de los distintos métodos durante el entrenamiento.

Finalmente, como resultado del entrenamiento se obtiene también la política aprendida, que se puede definir como una función que dado un estado observado del entorno, retorna la acción óptima que debe tomar el agente de acuerdo a lo aprendido durante el entrenamiento, y cuyo formato depende del método utilizado.

3.3. Controlador de elasticidad

Tanto los modelos o políticas resultantes del entrenamiento, como la política basada en umbrales utilizada como *baseline*, alimentan a un controlador de elasticidad. Este es un módulo de *Python* que, en base a la política seleccionada, se encarga de interactuar con el sistema simulado y controlar su elasticidad, recordando que la elasticidad de un sistema es su capacidad de adaptarse a la carga de trabajo minimizando el desperdicio de recursos.

Para la interacción con la simulación, el controlador se comunica por medio de una interfaz que provee operaciones que se encargan de implementar la interacción del agente con el sistema simulado. De esta manera, este módulo tiene la capacidad de observar el estado del entorno y agregar o quitar servidores.

Para poder ejecutar el controlador se necesita un archivo de configuración pasado por parámetro, similar al utilizado por los módulos de entrenamiento, el cual debe tener los mismos parámetros necesarios para realizar la simulación, pero también parámetros que definan qué política se quiere utilizar para controlar la elasticidad del sistema y dónde se encuentra, permitiendo además definir el tipo de *environment* a utilizar.

Luego, cada vez que se deba tomar una decisión el controlador pasará el estado del simulador como entrada a la política seleccionada y ésta devolverá la acción a realizar, la cual se ejecutará sobre el ambiente simulado.

Finalmente, este módulo genera un archivo de *log* y archivos CSV que contienen información sobre el rendimiento del sistema durante la simulación, permitiendo evaluar el funcionamiento del controlador con la política seleccionada.

3.4. Evaluación

Para la evaluación de las políticas se implementó un módulo de *Python* que procesa la información almacenada en los archivos CSV generados por el controlador de elasticidad. Este módulo genera métricas y gráficos útiles para la evaluación de las distintas políticas utilizadas.

Para la generación de las métricas y gráficos se tienen en cuenta la cantidad de servidores que están siendo utilizados y el tiempo de respuesta, datos que se obtienen de la simulación cada vez que el agente observa el entorno.

3.5. Interfaz

La interacción entre el simulador y el controlador se da a través de un módulo de *Python* que implementa un *PyEnvironment*. Para interactuar con MATLAB desde dicho módulo, se hizo uso de un motor de MATLAB para *Python*, a través de su correspondiente API (MathWorks, s.f.).

Un *PyEnvironment* es una clase abstracta que contiene operaciones especialmente diseñadas para ser utilizadas en un entorno de Aprendizaje por Refuerzo en *Python*. Estas operaciones permiten implementar la interacción usual entre un agente y el entorno, cuyo flujo se describe en el Algoritmo 6 y consiste en que el agente observe el estado del entorno, defina una acción, ejecute esa acción y observe el estado del entorno como resultado de esa acción.

Algoritmo 6 Flujo de interacción de un agente con su entorno

```
while training do
    actual_state ← env.getCurrentState()
    action ← policy(actual_state)
    new_state ← env.step(action)
end while
```

Algunas de las operaciones que permiten implementar dicho flujo se listan a continuación, siendo *TimeStep* una variable que almacena información sobre el entorno, incluyendo su estado y la recompensa asociada.

- *reset*: comienza una nueva secuencia y retorna el primer *TimeStep* de la secuencia.
- *current_time_step*: retorna el *TimeStep* actual.

- *step*: actualiza el entorno de acuerdo a la acción recibida y retorna un nuevo *TimeStep*.
- *observation_spec*: define el formato de las observaciones provistas por el entorno.
- *action_spec*: define el formato de las acciones provistas a *step*().

Cada implementación de la clase debe implementar la operación constructora, donde se definen los formatos de las acciones y las observaciones recibidas y provistas por el entorno. Dichos formatos luego son retornados al invocar a las operaciones *action_spec* y *observation_spec* respectivamente. Además, se debe redefinir la implementación de las operaciones *_step* y *_reset*, ya que son las que contienen la lógica particular del algoritmo y el entorno siendo utilizados.

La función *step* es la encargada de aplicar la acción recibida como parámetro al entorno, realizar una observación del entorno luego de aplicada la acción y, en base a dicha observación, calcular la recompensa obtenida como consecuencia de aplicar la acción. Por lo tanto, su implementación depende del entorno en el que se esté trabajando, en este caso, del simulador, y también de la función de recompensa utilizada para el cálculo de las recompensas. En ese sentido, el motor de MATLAB facilita la ejecución de las acciones en el entorno, y también la obtención de métricas del simulador que constituyen las observaciones antes mencionadas.

Por otro lado, la función *reset* es la encargada de volver a inicializar el entorno y obtener la primera observación del mismo. En este caso implica detener la simulación actual (si había una) e iniciar una nueva, eventualmente con nuevos parámetros de configuración que definan una nueva instancia de simulación. Nuevamente, el motor de MATLAB permite detener y reiniciar la simulación, así como también inicializar los nuevos parámetros de configuración.

En este caso, para poder implementar correctamente la interacción entre el agente y el entorno, es necesario que exista una sincronización entre el módulo de aprendizaje y la simulación. Dicha sincronización es importante sobretodo en la función *step*, ya que es la encargada de ejecutar una acción y retornar el estado del entorno luego de dicha acción.

Para ello se implementó el flujo que se describe en el Algoritmo 7, en donde *stopTime* es una constante del simulador que puede ser modificada desde fuera a través del motor de MATLAB y permite pausar automáticamente la simulación cuando el tiempo de simulación alcanza o supera el valor de esa constante. Por lo que el flujo consiste en ejecutar la acción correspondiente, actualizar el *stopTime* sumándole la cantidad de segundos de simulación que se desea correr antes de evaluar el resultado de ejecutar la acción, retomar la simulación, y luego quedarse esperando a que el tiempo de simulación haya alcanzado el tiempo de parada. Una vez que esto suceda, la simulación estará pausada, por lo que se procede a obtener el estado del simulador, a partir del cual se calcula y retorna la recompensa asociada a la acción que se ejecutó en el paso actual.

Algoritmo 7 Flujo de la función *step* con sincronización

Require: simulation paused
  Execute action
  Update stopTime ▷ Next simulation stop time
  Resume simulation
  while $simulationTime < stopTime$ **do**
    Wait s seconds
    Update simulationTime
  end while
  $state \leftarrow getState(simulation)$
  $reward \leftarrow computeReward(state)$
  return (state, reward)

Esta sincronización es indispensable para que haya coherencia en el cálculo de las recompensas asociadas a cada acción, ya que a través de ella se garantiza que transcurra siempre el mismo tiempo simulado entre que se ejecuta una acción y se miden las consecuencias de esa acción en el entorno. La implementación de esta sincronización pudo realizarse utilizando un conjunto de bloques dentro del simulador que permiten pausar automáticamente la simulación al alcanzar cierto tiempo, así como también el motor de MATLAB para interactuar con la simulación.

Finalmente, notar que algunos métodos admiten un espacio continuo de estados, mientras que otros solamente admiten un espacio discreto de estados, influyendo esto en el formato de las observaciones realizadas sobre el entorno. Por lo tanto, fue necesario implementar varios *PyEnvironments* para adaptar las características del espacio de estados al método de Aprendizaje por Refuerzo utilizado.

Capítulo 4

Simulación de un sistema HAS

Para la simulación del sistema HAS, se tomó como base el simulador descrito en el Apartado 2.3. Si bien la solución lograba representar una realidad del sistema modelado, esta contaba con algunas limitaciones que hacían que no se adaptara completamente a las necesidades del proyecto. En particular, para poder entrenar un algoritmo usando el simulador, este debería poder soportar gran cantidad de clientes, variar la carga de los mismos, y variar la cantidad de servidores disponibles: características que el simulador base no posee. Estas y otras limitaciones se detallan a continuación junto con los pasos seguidos para cubrirlas.

Por un lado, el simulador original modela la transmisión de un único video de duración infinita. Para que se asemeje más a un sistema real, en lugar de que los servidores ofrezcan un único video de duración infinita, debería haber videos con duraciones variables y finitas. Para implementar esto, se mantiene una lista de videos con sus respectivas duraciones, y en las solicitudes se hace referencia a estos datos. Con el pedido de un cliente asociado a una duración de video específica, el cliente mismo mantiene cuántos segmentos necesita solicitar para completar el video, y una vez alcanzada esa cantidad, deja de solicitar segmentos. Con este cambio se obtiene una carga más variable, ya que distintos clientes (y sus correspondientes solicitudes) existirán en el sistema durante un tiempo limitado de la simulación.

Otra limitante se da en que el sistema simulado cuenta con un único servidor, por lo que no tiene la capacidad de variar la cantidad de los mismos. Además, la implementación de los clientes no es escalable, estando limitada por la cantidad de bloques del simulador, ya que se debe colocar un bloque por cliente del sistema. En ese sentido, una de las mayores limitantes de la escalabilidad de la simulación es la cantidad de bloques máximos permitidos por la licencia,

por lo que la solución encontrada fue reducir la cantidad de bloques que se utilizan para modelar algunos componentes. En particular, se decidió cambiar la implementación para que todos los clientes pudieran modelarse como un solo bloque. La forma de lograr esto fue utilizando *Discrete Event Systems* (DES) para modelar el comportamiento de un conjunto de clientes. De esta forma se reemplazan los bloques de clientes por un bloque DES, que cumple el rol de planificador y permite simular una planificación de clientes, la cual consiste en un conjunto de clientes donde cada uno tiene asociado un identificador, un video a solicitar, y el segundo de simulación en el que comienza a solicitarlo. Esta implementación permite simular el comportamiento de una cantidad ilimitada de clientes, así como generar cargas que varíen a lo largo de la simulación.

Además, minimizar la cantidad de bloques necesaria para representar a los clientes permite agregar más servidores. Sin embargo, con múltiples servidores, surge la necesidad de contar con un balanceador de carga para repartir las solicitudes entre los servidores disponibles. También surge la posibilidad de habilitar y deshabilitar servidores, característica fundamental para el control de elasticidad del sistema. Esto se logró utilizando un bloque de tipo *Constant*, que representa la cantidad de servidores activos, y cuyo valor es consumido por la función que balancea la carga. El valor de este bloque tiene la capacidad de ser modificado desde un módulo externo al simulador, lo cual permite modificar la cantidad de servidores durante la simulación. Al modificarse la cantidad de servidores, se agrega un retraso temporal hasta que el cambio toma efecto en el sistema, reflejando así el tiempo que se tarda en agregar y quitar un servidor.

Cubrir estas limitantes permitió aumentar el rango de instancias representables por el simulador, enriqueciendo el proceso de aprendizaje del agente. En la sección 4.1 se describe en detalle la implementación del planificador de clientes mencionada anteriormente. Luego, en la sección 4.2 se describen la implementación y el criterio utilizado por el balanceador de carga, y en la sección 4.3 se presentan las métricas ofrecidas por el simulador. Finalmente, en la sección 4.4 se describe el módulo encargado de la generación de los archivos de configuración que definen las distintas instancias de simulación.

4.1. Implementación del planificador

El funcionamiento de los clientes en el planificador se basa en el prototipo descrito en la Sección 2.3, utilizando las estructuras y eventos proporcionados por *Discrete Event Systems*, descrito en la Sección 3.1. El bloque de planificador tiene una entrada y una salida, ambas son de tipo *Entity*, que representa cada solicitud: la entrada representando una respuesta a una solicitud que fue procesada o no por un servidor, y la salida representando una solicitud de transacción saliente hacia un servidor. Además, cada *Entity* cuenta con ciertos atributos que serán modificados durante la simulación para representar el estado de la solicitud.

El planificador cuenta con tres *Storages*: el *Storage Playback Buffers*, el *Storage Playback Servers*, y el *Storage Request Generators*. Los tres están implementados como una *Queue FIFO*, con distintas capacidades. En el caso de *Playback Buffers*, representa los *buffers* de reproducción de cada uno de los clientes, por lo que su capacidad será la cantidad de clientes multiplicada por la capacidad de los *buffers*. Por otro lado, *Playback Servers* representa el recurso que reproduce un segmento de video en cada uno de los clientes, pero solo va a reproducir un segmento por cliente a la vez, así que tiene capacidad igual a la cantidad de clientes. Finalmente, *Request Generators* tiene capacidad infinita, ya que es el *Storage* donde se generan las solicitudes de todos los clientes, y estas estarán determinadas por los eventos que se den en la simulación.

Al comenzar la simulación, se ejecuta el evento de *setup*, donde se inicializan las variables necesarias y se hace que cada uno de los clientes envíe su primer solicitud, lo cual comienza el flujo de la simulación. Esto último se hace creando eventos de generación de solicitudes en el *Storage Request Generators*. La creación de estos eventos se hace en base a la planificación de pedidos durante toda la simulación, que indica qué cliente hará cada pedido, sobre qué video, y en qué momento. En particular, cuando se crean los eventos de tipo *Generate*, se indica el *delay* que existe hasta que estos eventos efectivamente se realicen, y es con este valor que se puede preparar la generación de todos estos eventos al comienzo de la simulación, teniendo en cuenta que al inicio solamente se programa la solicitud del primer segmento de video para cada cliente, ya que el momento de solicitud de los siguientes segmentos quedará determinado por el comportamiento del sistema durante la simulación.

Cuando pase el tiempo indicado, se ejecutará un evento de tipo *Generate* en el *Storage Request Generators*. Aquí, se creará la solicitud, inicializando los atributos necesarios para el correcto funcionamiento de la simulación. Uno de esos atributos es la representación del video elegida, que se elige aplicando el algoritmo adaptativo. Luego, se utilizará el evento *Forward* para quitar la solicitud del *Storage* y enviarla a la salida del planificador, teniendo en cuenta el *inter-request delay* definido por el algoritmo adaptativo. En el Algoritmo 8 presentamos el algoritmo adaptativo utilizado en este proyecto, definiendo $dt = \frac{2}{3}n$ y $rt = \frac{1}{3}n$.

El flujo continúa cuando una solicitud ingresa al planificador. Debido a la configuración del mismo, todo lo que pase por el input será enviado al *Storage Playback Buffers*. Al ingresar una solicitud a este *Storage*, se ejecutará un evento de tipo *Entry*, en el que se realizarán las siguientes acciones:

Utilizando los atributos de la solicitud, se obtiene información del estado del servidor que atendió la solicitud. En el caso de que el servidor haya sido desconectado, se actualiza el estado interno que mantiene las conexiones activas de cada cliente, y se destruye la solicitud.

Si la solicitud no ha sido destruida, se obtienen los datos del segmento descargado para actualizar el estado interno de los pedidos por cliente. En el caso de que aún queden segmentos que solicitar para completar el video pedido por

Algoritmo 8 Algoritmo adaptativo

Input: n (number of segments on client's replay buffer)
Input: dt (inter-request delay threshold)
Input: rt (representation threshold)
Input: d (segment duration)
if $n \leq dt$ **then** ▷ Inter-request delay selection
    inter-request-delay = 0
else
    inter-request-delay = d
end if
if $n < \frac{rt}{2}$ **then** ▷ Representation selection
    downgrade one representation (if possible)
else if $\frac{rt}{2} \leq n \leq rt$ **then**
    keep current representation
else
    upgrade one representation (if possible)
end if

ese cliente, se genera un nuevo evento de tipo *Generate* en el *Storage Request Generators*. Luego, si el cliente no está 'reproduciendo' ningún segmento en ese momento, la solicitud se envía al *Storage Playback Server* mediante un evento de tipo *Forward*. En caso contrario, la solicitud es destruida. Esto es para no almacenar explícitamente todos los segmentos descargados, y en su lugar se mantiene en el estado interno del planificador cuántos segmentos hay almacenados en el *buffer* de cada cliente.

Existe otro evento cuando las solicitudes entran al *Storage Playback Server*. Aquí simplemente se actualiza el estado del *buffer* y del reproductor, para luego generar un evento de tipo *Timer*, al que se le pasa un *delay* que representa el tiempo de reproducción, y luego de ese *delay* se ejecuta el evento. Al momento de ejecutarse el evento, lo primero que ocurre es que se destruye la *Entity* asociada al segmento que se acaba de reproducir. Luego, si el cliente tiene una conexión activa, aún tiene segmentos de video para pedir y no tiene segmentos en el *buffer*, se genera una nueva solicitud en *Request Generators*. Si, en cambio, el cliente tiene segmentos almacenados en el *buffer*, va a crear un evento de tipo *Generate* en el *Playback Buffer*, que simplemente genera una entidad asociada a ese cliente y la envía al *Playback server* mediante un evento de tipo *Forward* para simular su reproducción.

De esta forma, mediante el conjunto de eventos mencionados se logra simular el comportamiento de una cantidad configurable de clientes que aplican el sistema HAS durante el flujo de *streaming*.

4.2. Balanceador de carga

Una vez que la solicitud abandona el cliente debe poder llegar a un servidor para ser atendida. Teniendo una cantidad variable de servidores, cada uno con una capacidad limitada, el total de solicitudes de los clientes debe dividirse entre los servidores disponibles. Esta tarea es realizada por un balanceador que distribuye las solicitudes basado en la carga actual de los servidores del sistema (es decir, la cantidad de solicitudes siendo procesadas por cada servidor).

Para ello, al momento de asignar una solicitud a uno de los servidores disponibles, el balanceador debe tener en cuenta múltiples características. En primer lugar, debe saber qué servidores están disponibles y cual es su carga de trabajo actual. Para eso, mantiene en su estado la carga de cada servidor y la cantidad de servidores activos. Cuando se desconecta un servidor, se desconecta el último, por lo que la cantidad de servidores activos es información suficiente para que el balanceador pueda asignar a las solicitudes un servidor disponible. Para hacer esto, busca en la lista de cargas de los servidores, el servidor con la menor carga, pero solo busca hasta la cantidad de servidores activos. Cuando se le asigne una solicitud a un servidor y cuando una solicitud termine de ser procesada por el servidor, se actualiza la carga en el estado del balanceador.

Otra tarea del balanceador de carga es establecer la conexión TCP con el cliente. Esta tarea solía pertenecerle al servidor en el simulador original, pero al aumentar la cantidad de servidores la conexión del cliente pasa a establecerse con el balanceador de carga, y se asume que ya hay una conexión establecida entre el balanceador y los servidores. Esta conexión se implementa de la misma manera que se implementaba en el servidor antes de modificarlo, lo que significa que existe una capacidad limitada de conexiones TCP que pueden estar siendo establecidas al mismo tiempo, por lo que se puede llegar a rechazar la solicitud de conexión por parte del cliente.

Además, para simplificar la simulación, se decidió que en el caso de que un servidor esté atendiendo una solicitud cuando es desconectado, el cliente asociado será desconectado y no enviará más solicitudes. Notar que realizar esta simplificación es equivalente a descartar las próximas solicitudes de dichos clientes de la planificación inicial, lo cual no afecta significativamente al entrenamiento.

4.3. Métricas

El simulador se construyó como una herramienta para entrenar una política de control de elasticidad que va a tomar decisiones sobre la asignación de recursos. Para poder tomar estas decisiones, el algoritmo debe poder acceder al estado del sistema. Es con ese objetivo que se implementó la extracción de métricas a ser consumidas por el algoritmo.

Las primeras métricas que se desean obtener son aquellas que indican el tiempo de respuesta de los servidores, ya que esta información será fundamental para evaluar el rendimiento de los servidores y la calidad de servicio ofrecida. Esta información se obtiene de la siguiente manera: cuando una solicitud es enviada por el planificador, se le inicializa un atributo que indica el tiempo de salida. Luego, una vez que el servidor atendió la solicitud, en el momento anterior a que regrese al planificador, se le inicializa un atributo que indica el tiempo de llegada. Con esos dos valores, se calcula el tiempo de respuesta para una solicitud, que se considera como el tiempo transcurrido entre que el cliente manda la solicitud y recibe la respuesta correspondiente.

Cada vez que se calcula el tiempo de respuesta de una solicitud, el simulador devuelve este valor (junto con datos que identifican a la solicitud y al cliente), y cada uno de estos datos alimentan un bloque que calcula el valor promedio del tiempo de respuesta general del sistema, tomando en cuenta una ventana de solicitudes limitada. Además, los datos de cada solicitud son retornados por el simulador, permitiendo calcular externamente el valor promedio del tiempo de respuesta para una ventana de tiempo limitada.

En el instante previo a que una solicitud exitosa regrese al planificador, se calculan múltiples métricas además del tiempo de respuesta. Una de ella es el *throughput*, o rendimiento, que indica la tasa de solicitudes o unidades de memoria procesadas correctamente por el sistema. Este cálculo se realiza manteniendo el total de solicitudes y de unidades de memoria procesadas por el sistema, permitiendo calcular ambos tipos de *throughtput* en cualquier punto de la simulación.

Otra métrica que se calcula en este momento de la simulación es la tasa de error. Una solicitud puede ser descartada por varios motivos: porque no puede establecer la conexión TCP con el balanceador de carga, porque el servidor no cuenta con la capacidad parar atender su solicitud, porque el servidor que atiende la solicitud es desconectado. Sea cual sea el motivo, se guarda la cantidad total de solicitudes descartadas, y se calcula la tasa de error dividiendo la cantidad total de solicitudes descartadas entre el total de solicitudes recibidas.

Además, se calcula la carga del sistema, medida como la cantidad de solicitudes siendo procesadas por el sistema. Finalmente, se mantiene la información de la cantidad de servidores activos, la cantidad de servidores esperados (teniendo en cuenta que puede haber servidores siendo activados o desactivados), y el tiempo actual de simulación.

En resumen, las métricas disponibles se listan a continuación:

- tiempo de respuesta promedio (para una ventana de solicitudes y para una ventana de tiempo)
- throughput (solicitudes y unidades de memoria transmitidas)
- tasa de error
- carga del sistema

- cantidad de servidores activos
- cantidad de servidores esperados
- tiempo actual de simulación

Estas métricas conforman el estado de la simulación, permitiendo realizar observaciones sobre el entorno que sirvan como base para la toma de decisiones al momento de controlar la elasticidad del sistema.

4.4. Generación de instancias

El simulador cuenta con un conjunto de variables configurables que definen las características de la instancia a ser simulada. Algunas de estas variables definen las características del sistema HAS siendo simulado y los videos disponibles, mientras que otras definen las características de los clientes HAS siendo simulados y la planificación de clientes a simular.

Estas variables se listan a continuación, y pueden ser inicializadas a partir de un archivo externo antes de ejecutar la simulación.

- *simultaneous_connections* (*default*=25): Cantidad máxima de conexiones TCP que pueden ser establecidas en simultáneo por el sistema.
- *RTT* (*default*=0.001): *Round trip time* en segundos, usado para definir el tiempo que toma el establecimiento de conexión TCP.
- *server_setup_delay* (*default*=8): Tiempo de encendido y apagado de servidores (en segundos).
- *initial_active_servers* (*default*=1): Cantidad inicial de servidores activos.
- *httpQueueCapacity* (*default*=200): Capacidad de cola de espera HTTP en cada servidor.
- *httpThreads* (*default*=60): cantidad de hilos HTTP disponibles para cada servidor.
- *fetch* (*default*=8000000): Velocidad de *fetch*, en unidades/segundo.
- *ioBuffers* (*default*=60): cantidad de I/O *Buffers* de cada servidor.
- *buffers_capacity* (*default*=15): capacidad de cada I/O *Buffer*, en unidades.
- *blockSize* (*default*=14600): tamaño de un bloque de archivo, en unidades.
- *drainTime* (*default*=0.00256): tiempo que tarda el servidor en drenar un bloque de archivo, en segundos.
- *videos* (*default*=3): cantidad de videos disponibles en el sistema.
- *duration* (*default*=2): duración de cada segmento de video, en segundos.

- *videos_durations* (*default*=[120, 600, 1200]): arreglo que contiene la duración total de cada video disponible, en segundos.
- *representations* (*default*=5): cantidad de representaciones disponibles para cada video.
- *representation_sizes*: matriz que en la posición i,j contiene el tamaño (en unidades) de un segmento del video i en la representación j. El valor por defecto para las representaciones es:
$$\begin{bmatrix} 128 & 256 & 512 & 1024 & 2048 \\ 63000 & 90000 & 146000 & 256000 & 483000 \\ 880000 & 1240000 & 2050000 & 1820000 & 3480000 \end{bmatrix}$$
- *schedules*: cantidad total de solicitudes de video en la planificación.
- *schedule*: planificación de clientes representada como un arreglo de tuplas, donde cada tupla es de la forma (id, video, segundo), siendo id el identificador del cliente, video el identificador del video a ser solicitado por el cliente, y segundo el segundo de simulación en el que dicho cliente comienza a solicitar dicho video.
- *clients*: cantidad total de clientes en la planificación. (Coincide con *schedules*)
- *playback_bufffer_capacity* (*default*=15): capacidad (en segmentos) del *buffer* de reproducción de cada cliente.
- *delay_threshold* (*default*=5): *threshold* que utiliza el algoritmo adaptativo en el cliente para definir el *inter-request delay*.
- *rep_threshold* (*default*=5): *threshold* que utiliza el algoritmo adaptativo en el cliente para definir la representación del próximo segmento.
- *default_rep* (*default*=3): representación por defecto.

Tanto las características del sistema HAS como las de los clientes HAS se mantuvieron fijas en sus valores por defecto durante la realización de este trabajo. Por lo que cada instancia queda determinada por la planificación de clientes.

Para generar las distintas instancias de entrenamiento y evaluación se implementó un *script* de *Python*, a partir de ahora llamado generador de instancias, que recibe: el nombre del archivo de configuración a ser generado, la duración en segundos de la simulación de la instancia (que debe ser múltiplo de 60) y un valor λ que es utilizado al generar la planificación de clientes.

El generador de instancias crea un archivo de extensión .m, que será el archivo de configuración a ser cargado por MATLAB para inicializar las variables del simulador. Este archivo contendrá la definición del tipo Entidad, con sus respectivos atributos, y además una línea por cada variable configurable de la forma *nombre = valor;*, siguiendo la sintaxis requerida por MATLAB. Todas las variables que definen características del sistema o cliente HAS se definen con

sus valores por defecto, mientras que la planificación de clientes se genera en base al parámetro de entrada λ.

Este parámetro indica la cantidad esperada de nuevos clientes por minuto de simulación. Notar que esto no define la cantidad de solicitudes por minuto que recibirá el sistema de *streaming*, ya que por el comportamiento de un sistema HAS, las solicitudes enviadas por cada cliente dependerán de como se desarrolle la simulación, por lo que solo se puede controlar en qué momento cada cliente envía la primera solicitud.

Entonces, a partir de la duración de la simulación se calculan los minutos de simulación, y luego, para cada minuto se sortea una cantidad de clientes siguiendo una distribución de *Poisson* de parámetro λ. Estos clientes luego se distribuyen de manera uniforme entre los segundos del minuto correspondiente. Esto permite determinar, para cada cliente, el segundo de simulación en el que realiza su primera solicitud. Finalmente, para cada cliente se elige aleatoriamente entre los 3 videos ofrecidos por el sistema, con pesos (10,20,1), asignándole mayor peso al video de media duración y calidad, y menor peso al video de mayor duración y calidad. De esta forma se construye la planificación, formada por tuplas de la forma (id, video, segundo), donde a cada cliente se le asigna un identificador entre 1 y la cantidad total de tuplas en la planificación.

De esta forma el generador de instancias crea un archivo de configuración para inicializar las variables del simulador, manteniendo fijas las características del sistema HAS y generando, de manera aleatoria, en base a la cantidad esperada de nuevos clientes por minuto, la planificación de clientes que define la instancia a ser simulada. Finalmente, el generador de instancias puede invocarse desde otros módulos, permitiendo generar automáticamente los archivos de configuración asociados a cada episodio durante el entrenamiento.

Capítulo 5

Implementación de estrategias de Aprendizaje por Refuerzo para control de elasticidad

En este capítulo se detalla el desarrollo del generador de políticas descrito en el Capítulo 3, profundizando en el proceso de implementación, entrenamiento y aplicación de distintos métodos de Aprendizaje por Refuerzo para el control de elasticidad del sistema de *streaming* simulado descrito en el Capítulo 4.

Se estudiaron tres métodos: *DQN*, SARSA y *Policy Gradient*, con el objetivo de evaluar el desempeño de distintas técnicas para la resolución del problema en estudio, evaluando también ventajas y desventajas del uso de cada método.

Recordando que el problema en estudio consiste en realizar un control de elasticidad horizontal de un sistema de *streaming* simulado, de manera tal de adaptar la cantidad de servidores disponibles en el sistema en base a la demanda, minimizando el desperdicio y manteniendo un nivel de rendimiento aceptable, definiendo en este caso un rendimiento aceptable como aquel que garantice una reproducción de videos sin interrupciones. En este caso, dado que se trabajó con un sistema HAS que secciona los vídeos en segmentos de dos segundos de duración, se garantiza una reproducción sin interrupciones manteniendo un tiempo de respuesta menor a dos segundos. Como además se busca minimizar el desperdicio, el objetivo es mantener un tiempo de respuesta inferior pero lo más cercano posible a dicho valor.

En ese sentido, se busca entrenar una política que en determinado momento defina una acción a tomar, en base a métricas calculadas a partir de observaciones

al estado del sistema. Siendo las acciones posibles: quitar un servidor, mantener la cantidad actual de servidores o agregar un servidor, y siendo el estado del sistema el conjunto de métricas de rendimiento disponibles en la simulación.

El entrenamiento se realiza en base a episodios, donde cada episodio está definido por una duración (en segundos) que indica el tiempo simulado asociado a ese episodio. Luego, la cantidad de iteraciones que tendrá cada episodio quedará determinada a partir del tiempo a transcurrir entre observaciones.

En todos los casos, se entrenó durante 1000 episodios y se evaluó durante 100 episodios, teniendo cada episodio una duración de 15 minutos y realizando observaciones cada diez segundos simulados, realizando entonces 90 iteraciones por episodio.

Por otro lado, para cada episodio se define una tasa de arribo de clientes, que es utilizada por el generador de instancias para generar un archivo de configuración para la simulación, incluyendo una planificación de clientes que siga esa tasa de arribos. Por ejemplo, si la tasa de arribos definida para un episodio es 20, entonces se generará una planificación de clientes en la que en cada minuto llegarán x clientes nuevos, siendo x un valor independiente para cada minuto simulado, muestreado según una distribución *Poisson* de parámetro $\lambda = 20$, y donde la llegada de dichos clientes se distribuirá de manera uniforme a lo largo del minuto, como fue previamente detallado en la Sección 4.4 del Capítulo 4.

En todos los casos se definieron tasas de arribo con valores entre 10 y 30, de manera de poder representar episodios con distintas cargas de trabajo que fueran manejables por el sistema simulado, teniendo en cuenta las características definidas para el mismo.

Finalmente, como la implementación de los distintos métodos fue un proceso iterativo e incremental, inicialmente se realizaban comparaciones en base la recompensa acumulada obtenida durante el entrenamiento, y luego se definieron métricas que permitieran comparar el rendimiento de las distintas políticas generadas. Dichas métricas consisten en el porcentaje de permanencia del sistema en una serie de estados durante la evaluación, dichos estados se listan a continuación y cada uno de ellos representa un cierto nivel de rendimiento del sistema.

- *waste*: este estado se da cuando el tiempo de respuesta promedio del sistema es menor a un segundo y el sistema está utilizando más de un servidor. Este estado refleja que el sistema podría estar desperdiciando recursos.
- *optimal*: este estado se da cuando el tiempo de respuesta promedio del sistema está entre uno y dos segundos, cuando el tiempo de respuesta promedio es menor a un segundo pero el sistema está utilizando la menor cantidad de servidores posible, o cuando el tiempo de respuesta promedio es mayor a dos pero el sistema está usando la mayor cantidad de servidores posible. Este estado refleja que el sistema se encuentra dando el rendimiento esperado, o de lo contrario, haciendo uso mínimo o máximo

de sus capacidades, lo cual no se considera desperdicio o mal rendimiento.

- *recoverable bad performance (rbp)*: este estado se da cuando el tiempo de respuesta promedio del sistema está entre dos y cuatro segundos y el sistema no está utilizando la mayor cantidad de servidores posible. Este estado refleja que el sistema se encuentra dando un rendimiento peor al esperado pero que aún es considerado recuperable, ya que puede haber tiempos de respuesta mayores a dos segundos sin que esto implique necesariamente una interrupción en la reproducción de vídeo, ya que los clientes pueden tener segmentos en su *buffer* de reproducción.
- *unrecoverable bad performance (ubp)*: este estado se da cuando el tiempo de respuesta promedio del sistema es mayor a cuatro segundos y el sistema no está utilizando la mayor cantidad de servidores posible. Este estado refleja que el sistema se encuentra dando un rendimiento peor al esperado y es poco probable que pueda recuperarse.

En base a la cantidad de iteraciones en que el sistema se encuentra en cada uno de estos estados durante la evaluación de una política puede evaluarse qué tan bien se está comportando bajo dicha política. Estas métricas se utilizaron tanto para la comparación entre políticas de un mismo método, como para la comparación entre políticas de los distintos métodos estudiados.

En las siguientes secciones se detalla el proceso de desarrollo de cada método, describiendo los pasos seguidos y las dificultades enfrentadas hasta obtener la mejor política alcanzada para cada método en el marco de este trabajo. En particular, no se pudo profundizar en algunos de los métodos estudiados debido a que los tiempos de entrenamiento elevados y la escasez de recursos dificultaron la completa exploración de los mismos dentro tiempo disponible para la realización del proyecto.

5.1. DQN

De acuerdo a lo mencionado en el Apartado 2.4.1, el método DQN incorpora dos elementos que ayudan a que el aprendizaje sea más eficaz: un mecanismo llamado *experience replay* y el uso de una segunda red neuronal. El *experience replay* tiene como objetivo eliminar correlaciones en la secuencia de observaciones y mantiene un *replay buffer* que almacena la trayectoria que sigue el agente a lo largo del entrenamiento. De esta manera, en cada paso que toma el agente se almacena un elemento que contiene el estado del cual se partió, la acción tomada y el estado al cual se llegó incluyendo la recompensa recibida, para luego, al momento de entrenar, tomar un *batch* de estos elementos. Se decidió inicializar el *replay buffer* antes de comenzar el entrenamiento con 1000 experiencias, que resultan de tomar acciones aleatorias, mientras que para el entrenamiento se definió un tamaño de *batch* de 32 elementos.

El uso de la segunda red neuronal tiene el objetivo de reducir las correlaciones entre los parámetros que actualmente tiene la red y los valores objetivos. La misma es actualizada periódicamente, a diferencia de la red principal que se actualiza en cada iteración. Esta segunda red se utiliza para actualizar los parámetros de la primera, logrando de esta manera que la predicción y el valor objetivo sean independientes. Se decidió actualizar la segunda red neuronal cada diez iteraciones.

Con respecto a la estructura de las redes neuronales, se definió un modelo de red de tipo *Sequential*, basada en la red presentada en (Bitsakos y cols., 2018), con una capa de entrada cuya dimensión se adapta a la observación del entorno, en este caso, recibe un subconjunto de las métricas de rendimiento obtenidas del simulador. El tipo de las capas es *Dense*, es decir, que cada neurona está conectada con todas las de la capa anterior y la siguiente. Se definieron tres capas ocultas de 64, 128 y 256 neuronas respectivamente, además de la capa de entrada y la capa de salida, esta última con tres neuronas, una para cada acción posible. Se utilizó la función de activación *ReLU*, usada por la red para transmitir información entre neuronas de distintas capas. Se definió utilizar el optimizador *Adam*, que tiene como objetivo optimizar los valores de los parámetros de la red con tal de reducir el error al entrenar.

Por otro lado, como se mencionó en la Sección 2.4 del Capítulo 2, un componente importante del entrenamiento es la decisión de, en cada paso que debe tomar el agente para interactuar con el ambiente, explotar el conocimiento adquirido hasta el momento o explorar otra acción que quizás no le dará una recompensa máxima inmediata pero que puede llevarlo a un mejor estado en el futuro. Se decidió seguir la segunda estrategia mencionada, es decir, que el agente comience explorando la mayor parte del tiempo y a medida que avanza el entrenamiento explorar cada vez menos. Esto en la practica se lleva a cabo fijando un número ϵ, el cual irá disminuyendo, y en cada paso el agente tomará la mejor acción posible de acuerdo a lo aprendido con probabilidad 1 - ϵ. De otra manera, tomará una acción aleatoria dentro de las posibles con probabilidad ϵ.

En ese sentido, experimentamos con dos opciones para ir variando el número ϵ. La primera de ellas fija el valor inicial en 1, otro valor que representa una tasa de decremento exponencial y un valor de ϵ mínimo para calcular el ϵ en cada paso como en el Algoritmo 9. De esta manera, el ϵ comenzará con un valor de 1 e irá decayendo gradualmente hasta llegar a 0.01.

Algoritmo 9 Primer método de decremento de *epsilon* tomado y adaptado de (PyLessons, s.f.)

Input: *current_episode*
$min \leftarrow 0{,}01$
$initial \leftarrow 1$
$decay \leftarrow -0{,}004$
$epsilon = min + (initial - min) * e^{decay * current\_episode}$

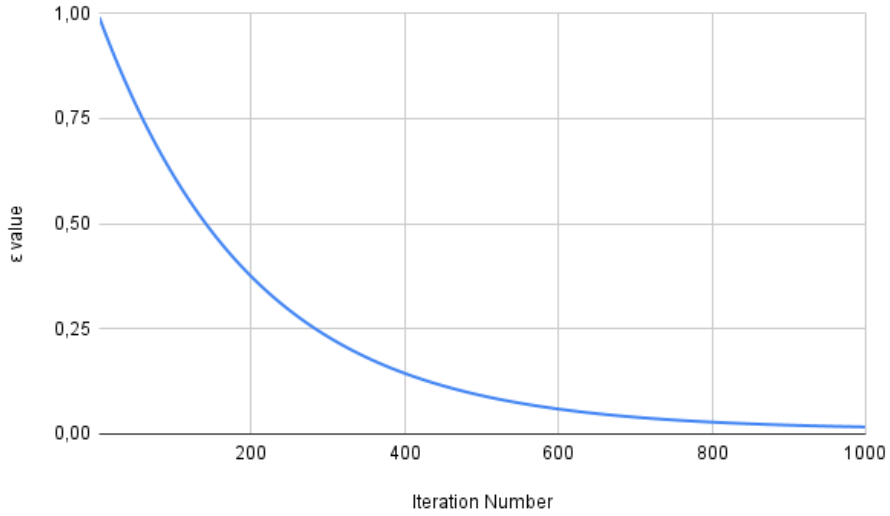

Figura 5.1: Gráfica de decremento de ϵ aplicando el Algoritmo 9

El comportamiento de este algoritmo puede observarse en la Figura 5.1.

Para la segunda opción, se buscó una función de decremento que mantenga un valor alto de ϵ por más iteraciones, obligando al algoritmo a explorar durante este tiempo. Para compensar, luego de la fase de exploración, el valor de ϵ debería decrementar rápidamente hasta acercarse a un valor mínimo. Se obtuvo este comportamiento con el Algoritmo 10, cuyo comportamiento se puede observar en la Figura 5.2

Algoritmo 10 Segundo método de decremento de *epsilon* tomado de (Subbiah Natarajan, s.f.)

Input: *current_episode*
Input: *total_episodes*
$min \leftarrow 0{,}01$
$A \leftarrow 0{,}5$
$B \leftarrow 0{,}1$
$C \leftarrow 0{,}1$
$standardized\_time \leftarrow (current\_episode\text{-}A^{*}total\_episodes)/(B^{*}total\_episodes)$
$cosh\_st \leftarrow cosh(e^{-standardized\_time})$
$epsilon{=}max(min, 1.1-(1/cosh\_st + (current\_episode * C/total\_episodes)))$

Uno de los grandes desafíos a la hora de implementar un algoritmo de DQN es la elección de la función de recompensa, ya que es en base a esto que el modelo

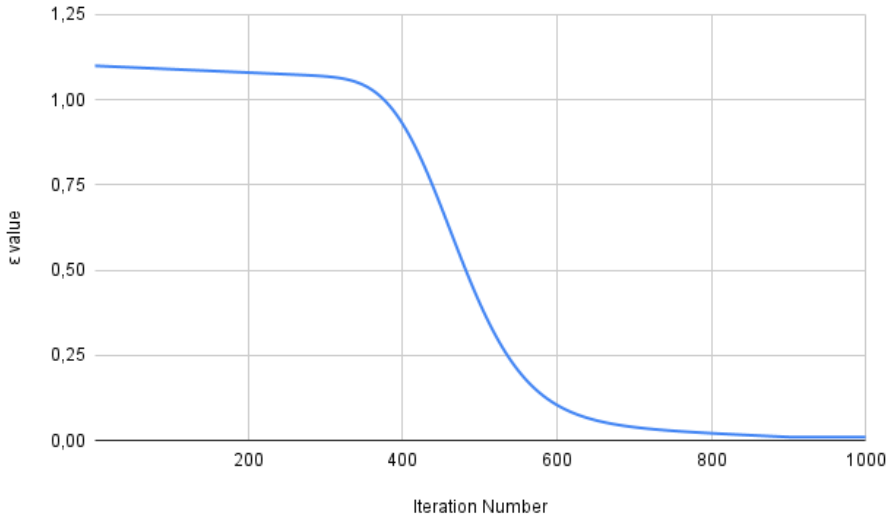

Figura 5.2: Gráfica de decremento de ϵ aplicando el Algoritmo 10

evaluará el estado y tomará decisiones. El proceso de ingeniería de recompensa fue iterativo. Para elegir el algoritmo de decremento con el que trabajar, al inicio se evaluaron ambas opciones utilizando una misma versión de la función de recompensa. En promedio, los valores de la recompensa acumulada durante el entrenamiento cuando se utilizaba el Algoritmo 9 tendían a ser mayores que cuando se utilizaba el Algoritmo 10. Teniendo esto en cuenta, se decidió seguir adelante con el Algoritmo 9. Una vez definido el algoritmo de decremento, se continuó con el proceso iterativo probando distintas combinaciones de funciones de recompensa que tenían en cuenta varias de las métricas obtenidas del entorno, asignando distintos pesos, que se fueron ajustando a medida que se observaban los resultados. A su vez se fueron descartando algunas métricas que en la práctica no influían en el resultado.

Las primeras versiones de la función de recompensa utilizaban las métricas de tiempo de respuesta, carga total del sistema y servidores activos, ya que fueron reconocidos como los datos que influenciarían la decisión de la política. En una primera etapa, se realizaron ejecuciones de entrenamiento con distintas funciones de recompensa, asignándole distintos pesos a cada una de las métricas, y variando también la recompensa que se asignaba para recompensar buenos estados y para penalizar estados malos.

A medida que se ajustaba la función de recompensa, se observó que la métrica de carga del sistema no aportaba información relevante que no estuviera ya implícita por el tiempo de respuesta y la cantidad de servidores activos. Por

estos motivos, y para simplificar la función de recompensa, se decidió quitar esta métrica.

Eventualmente, se encontraron algunos casos bordes que no se estaban teniendo en cuenta, en particular:

- Cuando el tiempo de respuesta se encontraba entre cero y un segundo, pero se estaba usando el mínimo de servidores disponibles.
- Cuando el tiempo de respuesta era mayor a dos segundos, pero se estaba usando el máximo de servidores disponibles.

Estos casos representan un estado en el que el rendimiento del sistema no es óptimo, pero está haciendo lo mejor que puede con los recursos disponibles. Como no hay una decisión que pueda tomar el agente de manera que mejore el estado del sistema en ese punto, se decidió dar un valor de recompensa positivo en estos casos en vez de uno negativo, como se estaba haciendo en iteraciones anteriores, antes de reconocer estas situaciones. Con esto en mente se definieron además las métricas utilizadas para la evaluación. Luego, las siguientes versiones de funciones de recompensa se basan en identificar en cuál de los cuatro rangos definidos al comienzo de este capítulo se encuentra el sistema de acuerdo al tiempo de respuesta y servidores activos, tomando en cuenta los casos bordes, pero variando los valores de recompensa que se suman o restan en diferentes rangos. Estas funciones se detallan a continuación, asumiendo que:

- rt = tiempo de respuesta promedio observado (para las solicitudes procesadas en los últimos diez segundos de tiempo simulado)
- s = cantidad de servidores activos
- maxS = máxima cantidad de servidores disponibles en el sistema
- minS = mínima cantidad de servidores disponibles en el sistema

La primera de ellas retorna una recompensa de 1 si se está en el estado *optimal* y negativo en el resto. En el caso de estar en el estado *rbp* se retorna una recompensa de -0.5, mientras que en el estado *ubp* se retorna -1, penalizando más fuertemente este caso. Por último, para el caso en que se está en un estado de *waste* se retorna una recompensa de $\frac{-s}{maxS}$. Se nota que en este caso R1(rt, s) $\in$ [-1, -0,13] siendo los extremos -1 cuando s = maxS, es decir, cuando se está usando el máximo de servidores disponibles y -0.13 cuando se está usando un solo servidor por encima de la cantidad mínima disponible. Con esta división en el estado *waste* se busca penalizar más cuanto más servidores se están utilizando.

$$R1(rt, s) = \begin{cases} \frac{-s}{maxS} & \text{if rt} \in [0,1) \text{ and s} > \text{minS} \\ +1 & \text{if rt} \in [1,\ 2] \\ -0{,}5 & \text{if rt} \in (2,\ 4] \text{ and s} < \text{maxS} \\ -1 & \text{if rt} \in (4,\ \text{ínf}] \text{ and s} < \text{maxS} \\ +1 & otherwise \end{cases}$$

La segunda función retorna los mismos valores de recompensa que R1 para los estados *rbp* y *ubp*, variándolo para los estados *optimal* y *waste*. Para el primero, en vez de recompensar con +1 como se hace en R1, se lo hace con +10, haciendo más énfasis en recompensar este estado, ya que se observó que la primera no valoraba las recompensas obtenidas en ese estado, manteniéndose en estados de penalización. Por otro lado, para el estado *waste* se penaliza con -1, no teniendo en cuenta la cantidad de servidores que se están utilizando, a diferencia de R1, ya que se observó que la proporción de servidores siendo utilizados no necesariamente es un indicador del desperdicio del sistema. De esta manera, se obtiene una función que para cualquier estado en que se encuentra el sistema, devuelve un valor constante, no dependiendo de parámetros.

$$R2(rt, s) = \begin{cases} -1 & \text{if rt} \in [0,1) \text{ and s} > \text{minS} \\ +10 & \text{if rt} \in [1,\ 2] \\ -0{,}5 & \text{if rt} \in (2,\ 4] \text{ and s} < \text{maxS} \\ -1 & \text{if rt} \in (4,\ \text{ínf}] \text{ and s} < \text{maxS} \\ +10 & otherwise \end{cases}$$

Por último, la tercera de ellas es muy similar a R2, retornando la misma recompensa para los estados *waste*, *rbp* y *ubp*. La única diferencia se da en el estado *optimal*, que se recompensa con +1 al igual que se hace en la función R1, ya que notamos que en R2 la recompensa del estado *optimal* era tan alto con respecto a los demás que permitía al agente pasar mucho tiempo en estados subóptimos sin notar una baja considerable en su recompensa acumulada, y por ende, no lograba enfocar el aprendizaje en evitar dichos estados.

$$R3(rt, s) = \begin{cases} -1 & \text{if rt} \in [0,1) \text{ and s} > \text{minS} \\ +1 & \text{if rt} \in [1,\ 2] \\ -0{,}5 & \text{if rt} \in (2,\ 4] \text{ and s} < \text{maxS} \\ -1 & \text{if rt} \in (4,\ \text{ínf}] \text{ and s} < \text{maxS} \\ +1 & otherwise \end{cases}$$

Estas funciones fueron evaluadas para encontrar cuál generaba una mejor política para el rendimiento del sistema, pero cada una ya representa una mejora con respecto a las iteraciones anteriores. Las razones de esto son varias, comenzando por lo observado anteriormente sobre los casos bordes. Además de estos casos, las funciones también tienen en cuenta estados de "recuperación", es decir: momentos en los que el estado es considerado "malo", pero no por decisiones de la política, sino por un cambio externo al cual aún no ha tenido la oportunidad de reaccionar, por ejemplo la subida drástica de la carga del sistema. Al reconocer estos estados y no penalizarlos tan fuertemente como en casos de malas decisiones, se evita que el agente aprenda de eventos que no necesariamente están relacionados.

Otro factor de mejora para estas nuevas funciones fue reducir la diferencia entre recompensas positivas y negativas. Iteraciones anteriores de la función tenían valores de recompensa muy altos o bajos, que daban un peso extra a ciertos comportamientos, provocando que, por ejemplo, muchas acciones con recompensa positivo tuvieran el mismo peso que una acción con recompensa negativo.

Por otro lado, también es importante la forma en la que se expresan las funciones de recompensa. Las primeras versiones de la función de recompensa buscaban minimizar la distancia a un tiempo de respuesta promedio objetivo, estrategia que no permitía diferenciar entre desperdicio de recursos y mal rendimiento, dificultando el correcto aprendizaje de las acciones a tomar en cada caso.

Otra mejora fue evitar incluir tanto el tiempo de respuesta como el desperdicio en la función de recompensa, asignándole distintos pesos en un intento de que el agente detecte qué acciones maximizan cada componente de la función. Dicho enfoque generaba políticas extremas, que tienden a enfocar sus decisiones en maximizar o minimizar una parte de la función de recompensa, es decir, mantener el tiempo de respuesta bajo asumiendo el costo de desperdiciar recursos, o el caso contrario. Por lo tanto, a la hora de construir una función de recompensa correcta, se pasó de buscar minimizar el tiempo de respuesta y el desperdicio de recursos, a intentar alcanzar un tiempo de respuesta objetivo minimizando el desperdicio. El primer enfoque era incorrecto, ya que el agente siempre buscaba minimizar el tiempo de respuesta e ignoraba el desperdicio, porque el desperdicio está relacionado con lo que se considera buen rendimiento, es decir, lo desperdiciado va a ser todo lo que se use y no sea necesario para alcanzar ese rendimiento. Si el objetivo es simplemente minimizar el tiempo de respuesta, siempre se puede mejorar el rendimiento y por lo tanto nunca se identifica un desperdicio.

Todas estas mejoras a la función de recompensa fueron resultado de análisis y puesta en práctica de las diferentes funciones, dando como resultado las funciones resultantes R1, R2 y R3. Aún así, si bien estas funciones representaron una mejora con respecto a sus antecesoras, su rendimiento aún dejaba mucho que desear. Ante esta situación, se identificó una posible problemática, que no estaba relacionada con la construcción de la función de recompensa, sino con el modelado del problema: la cantidad de posibles estados. Al mantenerse un espacio de estados continuo, en particular en cuanto al tiempo de respuesta, la cantidad de posibles estados era demasiado grande para ser explorado en su completitud durante el tiempo de entrenamiento. A raíz de esta problemática, existía la posibilidad de que la política entrenada se encuentre contra estados que nunca experimentó en su entrenamiento, y no sea capaz de reconocer estados similares. Para evitar esto se decidió cambiar el espacio de estados, surgiendo en la creación de un entorno discreto.

Para lograr discretizar el tiempo de respuesta se definieron cuatro rangos de acuerdo a su valor:

- rango 0: tiempo de respuesta promedio $\in [0,1)$

- rango 1: tiempo de respuesta promedio $\in$ [1,2]
- rango 2: tiempo de respuesta promedio $\in$ (2,4]
- rango 3: tiempo de respuesta promedio $\in$ (4, ínf]

Este nuevo entorno tiene la particularidad de que todas las métricas que se usan para identificar el estado son discretas: la cantidad de servidores activos y el rango del tiempo de respuesta promedio, resultando en un espacio de estados posibles finito, y mucho más reducido. De esta manera, es posible que durante el entrenamiento se llegue a explorar todos los estados, y sino, estados similares. Si bien en aplicaciones en las que se necesita un nivel fino de granularidad, la discretización podría resultar en la pérdida de información importante, en este caso la precisión que se está perdiendo no aporta grandes cantidades de información. Esto es porque que al buscar mantener el tiempo de respuesta promedio menor a 2 segundos, es suficiente con poder distinguir entre rangos. Por otro lado, se podría estar perdiendo información desde la perspectiva de la minimización del desperdicio de recursos, ya que en ese caso podría ser de utilidad distinguir qué tanto se está desperdiciando en base a qué tan cerca se está del objetivo. En este caso, no se buscaba hilar tan fino, por lo que se decidió que las ventajas de disminuir considerablemente el espacio de estados compensaban esta pérdida de información dada por la disminución de granularidad.

Al entrenar algoritmos en este nuevo entorno, con las funciones de recompensa antes mencionadas, se detectó una mejora considerable, que coincide con lo estimado. Por lo tanto, se decidió de allí en adelante utilizar el entorno discreto para entrenar los algoritmos. En la Figura 5.3 se puede observar el rendimiento al evaluar cada una de estas funciones utilizando dicho entorno.

Como se mencionó previamente, la evaluación se hizo sobre 100 episodios de 90 iteraciones cada uno, por lo que resulta en 9000 iteraciones en el total de la evaluación. De esta manera, se puede observar que la política que resultó de entrenar con la función R3 es la que pasa más iteraciones en el estado deseado *optimal*, logrando estar 7918 iteraciones allí, es decir, un 87,98 % del tiempo. Por otra parte, R1 le sigue en el orden, logrando estar 7353 iteraciones en *optimal*, lo que resulta en un 81,7 % del tiempo. Por último, R2 logra estar un 6847 de iteraciones en este estado, que equivale a pasar un 76,08 % del tiempo de evaluación en el estado deseado.

Si consideramos el desperdicio que tienen cada una de las políticas resultantes, medido por la cantidad de iteraciones que están en el estado *waste*, se visualiza que R3 logró una diferencia positiva entre las funciones. La misma estuvo solo 116 iteraciones en un estado de desperdicio de recursos, mientras que R1 estuvo un total de 517 iteraciones y R2 1603, lo cuál resulta en que esta última función logró el peor rendimiento entre las funciones en este estado, al igual que en *optimal*. Estos números equivalen a que R3 estuvo solo un 1,29 % del tiempo desperdiciando recursos, R1 un 5,74 % y R2 17,81 %.

Pasando ahora al estado *recoverable bad performance (rbp)*, se visualiza que

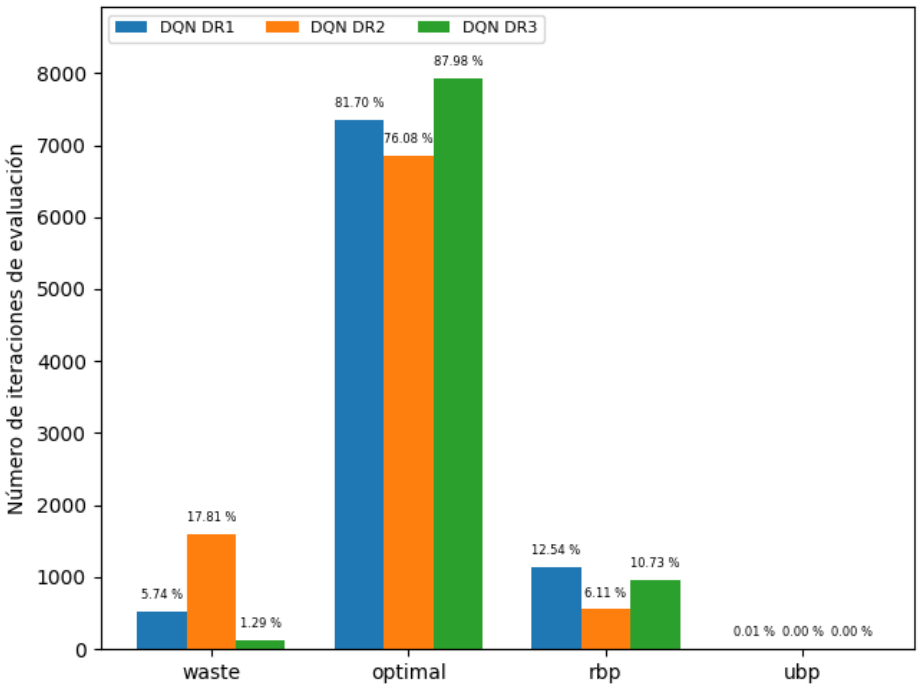

Figura 5.3: Comparación de políticas entrenadas utilizando el método DQN sobre un espacio de estados discreto con funciones de recompensa R1, R2 y R3 (DR1, DR2 y DR3 respectivamente).

la función que generó una política tal que se esté el menor tiempo posible en este estado es R2, seguida de R3 y resultando ser R1 la peor en rendimiento. En total, R2 pasa un total de 550 iteraciones en *rbp* (6,11 %), R3 está 966 iteraciones (10,73 %), mientras que R1 1129 iteraciones (12,54 %).

Por último, un aspecto importante a resaltar es que todas las funciones lograron un muy buen rendimiento si tenemos en cuenta el estado menos deseado *unrecoverable bad performan ce (ubp)*. Mientras que R2 y R3 no llegaron nunca a estar en este estado, R1 solo estuvo una iteración allí, lo que indica que se pudo recuperar rápidamente, por lo que se puede decir que no afecta la evaluación de esta política generada.

De esta comparación, se concluye que la política generada a partir del entrenamiento utilizando la función R3 es la de mejor rendimiento. La misma supera notoriamente a las otras políticas al pasar más tiempo en un estado óptimo y menos tiempo en estado de desperdicio. En comparación, en el estado *optimal* pasa más de un 6 % del tiempo que la entrenada con R1 y más de un 11 % que la entrenada con R2. En el estado *waste*, la diferencia a favor de R3 es de más

de 4 % con respecto a R1 y más de 16 % sobre R2. Ahora, si bien la política entrenada con R2 pasa menos tiempo que R3 en el estado *rbp*, esto no compensa la diferencia que hay entre las dos en los otros estados mencionados. Por lo dicho, se decidió seguir utilizando la función R3 para entrenar políticas con los otros métodos.

Como verificación de que generar un entorno discreto ayuda efectivamente a mejorar el entrenamiento de agentes para lograr una política con un mejor desempeño, se decidió hacer una comparación entre los entornos continuo y discreto con políticas generadas a partir de la función R3. En la Imagen 5.4 se puede apreciar la importante diferencia en rendimiento entre uno y otro. Si bien se visualiza que la política generada con un entorno continuo no desperdicia recursos en ningún momento de la evaluación, la misma pasa casi la mitad de las iteraciones (4444) con un tiempo de respuesta promedio muy elevado, lo que equivale a estar un 49,38 % en el estado *ubp*. Considerando el estado *optimal*, el entorno discreto generó un política que supera en 5010 iteraciones al continuo en la permanencia en este estado durante la evaluación, mientras el primero pasó un 87,98 % del tiempo en este estado, el segundo solo un 32,31 %. Por último, también en el estado *rbp* existe una diferencia a favor del entorno discreto, que permanece un 10,73 % del tiempo allí contra 18,31 % del entorno continuo. Por lo tanto, los resultados obtenidos permiten confirmar que el uso de un entorno discreto permitió obtener una política de control de elasticidad con un mejor rendimiento al ser aplicada sobre el entorno simulado.

5.2. SARSA

Como se mencionó en el Apartado 2.4.2, para entrenar a un agente este método se apoya en un función de calidad Q que depende del estado y la acción. De esta manera, de acuerdo a las métricas que se tienen en cuenta en la observación del entorno, se obtienen los estados y para cada estado se tiene un conjunto de acciones posibles. Esta función de calidad Q puede ser entonces implementada como una tabla y para cada combinación estado-acción se almacena un valor que representa qué tan bueno es tomar dicha acción si se está en ese estado. Luego, en cada paso que toma el agente y de acuerdo a la recompensa que recibe, se actualiza el valor correspondiente en la tabla Q como se muestra en la Ecuación 2.5.

Las métricas que se tuvieron en cuenta para representar el estado del sistema en este método son: cantidad de servidores activos, el tiempo de respuesta promedio discretizado de acuerdo a lo comentado en la sección anterior y la carga del sistema. Para la carga del sistema, y de acuerdo a los parámetros que se manejaron en la simulación, se asumió que la misma no va a superar nunca los 1000. Teniendo en cuenta lo anterior, además de que el sistema puede estar en cuatro diferentes tiempos de respuesta debido a la discretización, que la cantidad de servidores activos varía entre 1 y 15, y que hay tres acciones posibles, entonces

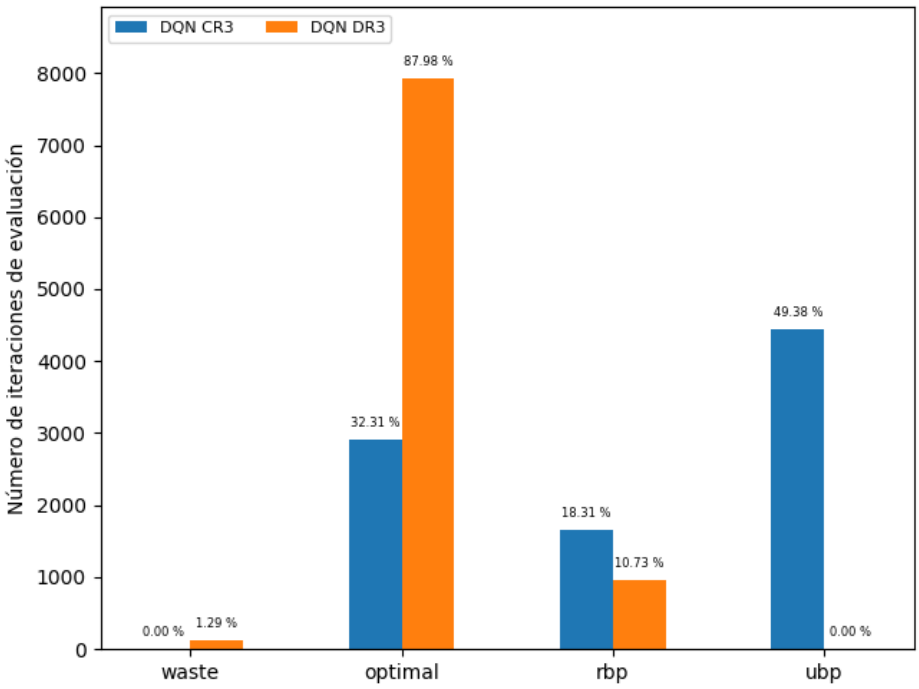

Figura 5.4: Comparación de políticas entrenadas utilizando el método DQN sobre espacio de estados continuo y discreto con función de recompensa R3 (CR3 y DR3 respectivamente).

la tabla Q tiene un tamaño de 180000. La misma se inicializó en 0 para cada par estado-acción antes de comenzar el entrenamiento.

Luego de entrenar con este método utilizando la función de recompensa R3, se evaluó la misma de acuerdo a lo detallado en la introducción de este capítulo y teniendo en cuenta que los resultados obtenidos no fueron satisfactorios, se experimentó en utilizar *experience replay*, al igual que en el método DQN. Al evaluar también la política generada al entrenar utilizando *experience replay*, no se pudo apreciar un gran cambio en el rendimiento. En la Figura 5.5 se observa el resultado de la evaluación de ambas políticas.

Si se observa el estado deseado *optimal*, ambas políticas permanecen un total de iteraciones bastante similar allí, siendo que la política generada al entrenar sin *experience replay* estuvo 4006 iteraciones, la otra estuvo 4028, lo que equivale a 44,51 % y 44,76 % del tiempo respectivamente. El estado donde se produce la mayor diferencia, y es en favor a la que utilizó *experience replay*, es en el *waste*. En este estado, la primera permaneció un total de 874 iteraciones (9,71 %) mientras que la segunda un total de 522 (5.8 %), lo que indica que agregar un

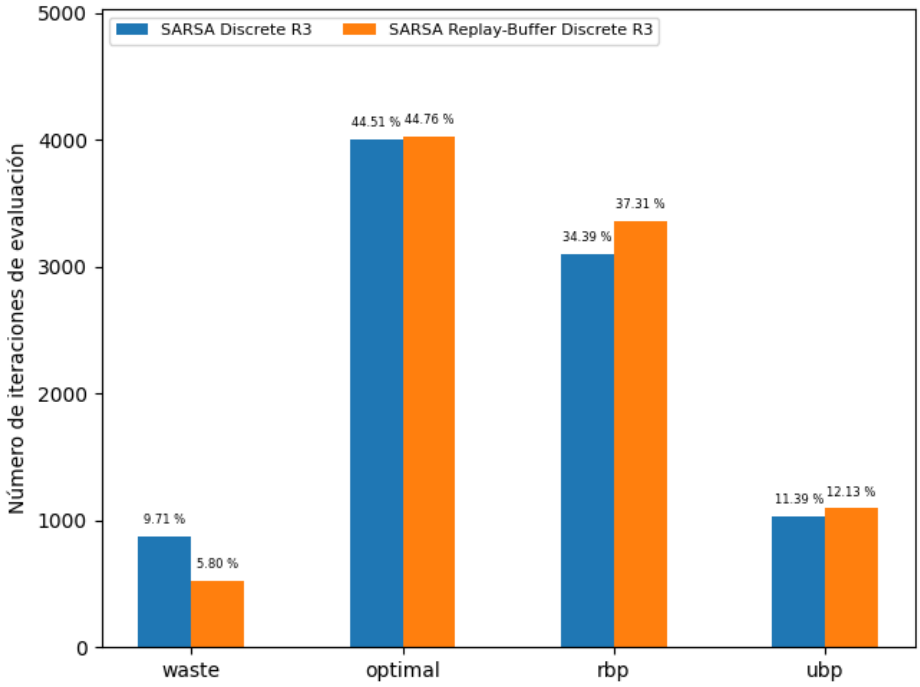

Figura 5.5: Comparación de políticas entrenadas utilizando el método SARSA sobre un espacio de estados discreto con función de recompensa R3, entrenado con y sin replay-buffer.

replay buffer al entrenamiento ayudó a que se desperdicien menos recursos. Como contraparte, en el estado *rbp*, la política entrenada sin *experience replay* pasó menos iteraciones que la otra. En este caso, la primera estuvo un total de 3095 iteraciones (34,39 %) contra 3358 (37,31 %). Por último, en el estado menos deseado, *ubp*, ambas políticas vuelven a tener un rendimiento parecido, ya que la que no utilizó *experience replay* pasó un total de 1025 iteraciones, mientras que la que sí utilizó, estuvo un total de 1092, que se traduce a pasar 11,39 % y 12,13 % del tiempo respectivamente en un estado de no recuperación.

Los resultados obtenidos no presentan evidencia suficiente para concluir que una política sea considerablemente mejor que la otra, de todas formas, de aquí en más se utilizará la política entrenada con *experience replay* como la mejor política alcanzada para este método, debido a que es la que permanece más tiempo en el estado *optimal* y menos tiempo en el estado *waste*.

5.3. Policy Gradient

Habiendo evaluado un método que no utiliza red neuronal, se decidió buscar un método para reutilizar la estructura de la red y la función de recompensa que dieron buen resultado con el método de DQN. Se eligió el método de Policy Gradient, que difiere en su comportamiento con los métodos anteriores, pero comparte con DQN el uso de la red.

Para la implementación nos basamos en (Gerón, 2019), y utilizamos la siguiente función de pérdida para entrenar a la red neuronal, basada en (Sutton y Barto, 2018):

$$loss(action\_prob, reward) = \begin{cases} -\log(action\_prob + 1e^{-5}) & if\ action\_prob = 0 \\ -\log(action\_prob) & otherwise \end{cases}$$

Donde *action_prob* es la probabilidad de la acción seleccionada.

En este caso se realizó una manipulación explícita de la red, la cual posee una entrada por cada métrica considerada (servidores activos, tiempo de respuesta promedio y carga del sistema) y una salida por cada acción posible (agregar un servidor, mantener la cantidad actual de servidores, quitar un servidor) llamada *logit* que representa para cada acción qué tan probable es tomarla partiendo desde cierto estado. Luego, a la salida de la red se utiliza la función *softmax* para construir una distribución de probabilidad teniendo en cuenta las acciones posibles desde el estado actual, y en base a esa distribución de probabilidad se sortea la acción a tomar.

Para asegurar estabilidad numérica, antes de aplicar la función *softmax*, se toma el máximo valor del vector de *logits* y se le resta ese valor a cada componente. Además, durante el cálculo de gradientes, se decidió normalizar el vector resultado que contiene los valores de los parámetros de la red antes de calcular las probabilidades, y también normalizar las recompensas obtenidas en cada episodio. Esto se realiza con el objetivo de evitar que un valor se vuelva mucho más grande que el resto, resultando en valores que superan lo que puede computar una máquina.

Al evaluar este método con el mismo tiempo de entrenamiento que los otros métodos, el rendimiento observado no fue satisfactorio. Si bien el entrenamiento resultó en una política que no desperdicia recursos, muchas de las solicitudes no llegaron a ser atendidas a tiempo debido al alto tiempo de respuesta. En el total de la evaluación, el sistema nunca se encontró en el estado *waste*, mientras que se pasó casi la mitad del tiempo (49,91 %) en el estado no recuperable *ubp*. En cambio, en el estado *optimal* estuvo un total de 2874 iteraciones, lo que equivale a pasar un 31,93 % del tiempo evaluado en el estado deseado. Por último, el sistema se encontró un total de 18,16 % del tiempo en el estado *rbp*, lo que equivale a 1634 iteraciones.

Observando el proceso de aprendizaje, se notó que este comportamiento era adquirido rápidamente, generando la sospecha de que el método mantenía un óptimo local tan pronto lo encontraba. Por esto, se intentó replicar el funcionamiento del *epsilon greedy* visto en el Algortimo 9 en este método, para generar un elemento de aleatoriedad en la política y forzarlo a explorar más estados.

Los resultados se muestran en la Figura 5.6, puede observarse que la versión del método con *epsilon greedy* tiene un comportamiento opuesto que la versión anterior, pero aún así no consigue resolver el problema. Se podría decir que su rendimiento es peor, ya que se encuentra casi enteramente en el estado *waste*, pasando allí un total de 8716 iteraciones, lo que equivale a 96,84 % del tiempo de evaluación. El sistema solo estuvo 277 iteraciones (3,08 % del tiempo) en el estado deseado *optimal*, mientras que el resto, solo 7 iteraciones (0.08 %), se pasó en *rbp*. Analizando los archivos de *log* se observó que al igual que con la política original, una de las probabilidades de tomar una acción se tornaba 1 rápidamente durante el entrenamiento, concluyendo que añadir exploración no permitió contrarrestar los efectos de este suceso, no mejorando el rendimiento de la política generada.

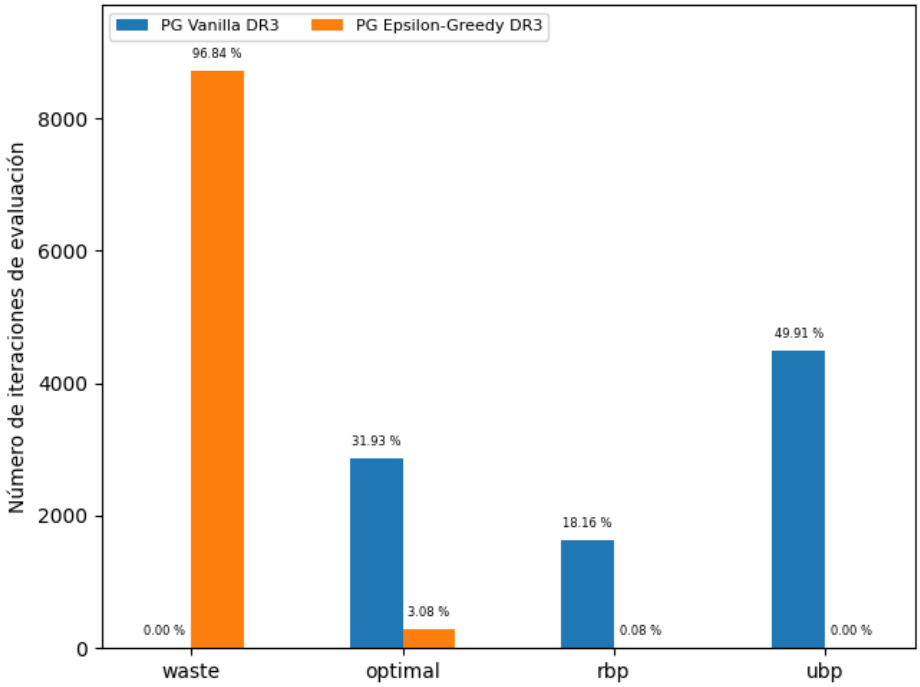

Figura 5.6: Comparación de políticas entrenadas utilizando el método Policy Gradient sobre un espacio de estados discreto con función de recompensa R3, entrenado con y sin *epsilon greedy*.

Capítulo 6

Evaluación

En este capítulo se realiza una comparación y evaluación de las políticas obtenidas a partir del proceso presentado en el Capítulo 5.

Para ello, se implementó como *baseline* una política de control de elasticidad que sigue una heurística basada en umbrales, y sirve como base para la evaluación de los distintos métodos, de forma de poder concluir si cada método es o no adecuado para atacar el problema estudiado. La misma consiste en observar el tiempo de respuesta promedio y tomar una acción de acuerdo a ello. Si el tiempo de respuesta se encuentra por debajo de un segundo y el sistema no está utilizando el mínimo de servidores disponibles, se toma la acción de quitar un servidor activo. Si el tiempo de respuesta se encuentra por encima de dos segundos y el sistema no está utilizando el máximo de servidores disponibles, se decide agregar un servidor más. De otra manera, se mantiene la cantidad de servidores activos hasta observar nuevamente el estado del entorno.

Esta política sigue la lógica de que si el tiempo de respuesta promedio del sistema se mantiene muy bajo durante las iteraciones más recientes, puede ocurrir que algunos de los servidores utilizados no sean estrictamente necesarios, por lo que se puede prescindir de uno de ellos. Por el contrario, si se observa que el tiempo de respuesta promedio es elevado, significa que hacen falta recursos para satisfacer la demanda, por lo que se debe agregar un servidor más. Por último, si el sistema se encuentra en un estado estable, no se requiere tomar ninguna acción.

Habiendo analizado ya todos los resultados obtenidos en cada uno de los métodos que se utilizaron para entrenar agentes de Aprendizaje por Refuerzo, se presenta en la Figura 6.1 la comparación de las mejores políticas obtenidas para cada uno de ellos, incluyendo la política *baseline*. Estas son: DQN en entorno discreto, SARSA utilizando *replay buffer* y Policy Gradient sin utilizar *epsilon greedy*. Todas estas se entrenaron con la función de *reward* R3.

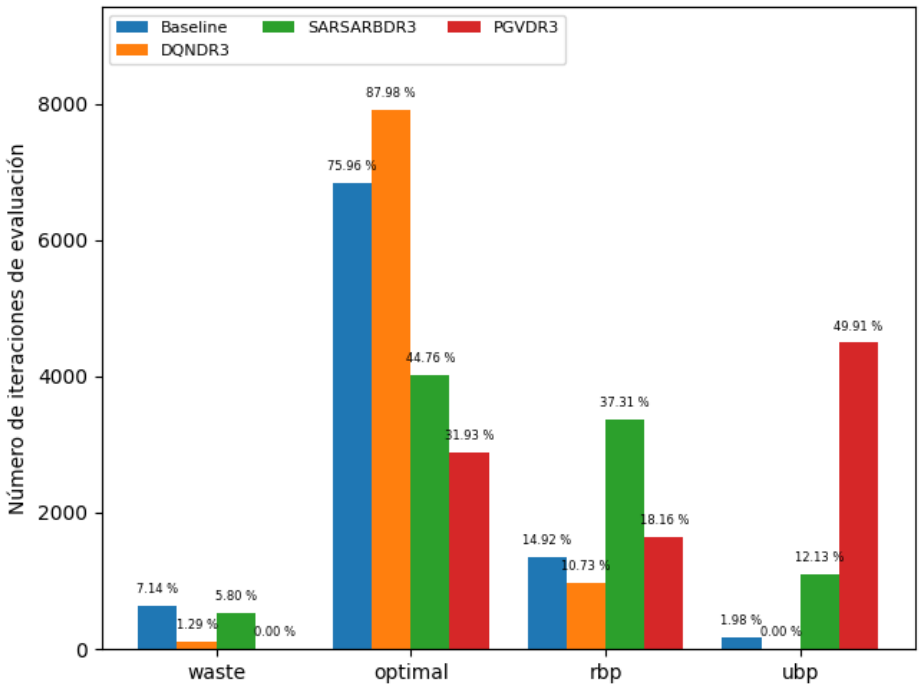

Figura 6.1: Comparación entre heurística basada en umbrales y mejores políticas obtenidas para los métodos DQN, SARSA y Policy Gradient.

El rendimiento del *baseline* por sí solo es relativamente bueno, manteniéndose un 75,96 % del tiempo de evaluación en el estado *optimal*, pero aún así tiene un 7,14 % del tiempo en *waste*, 14,92 % en *rbp* y 1,98 % en *ubp*. Se puede observar que el método DQN resuelve adecuadamente el problema de control de elasticidad, mientras que los métodos SARSA y Policy Gradient no cumplen el objetivo, incluso no logran superar el rendimiento de la política *baseline*. En el único aspecto que es superada la política de DQN, es que Policy Gradient no desperdicia servidores en ningún momento de la evaluación. Aún así, como se mencionó anteriormente, DQN solo pasa un 1,29 % es el estado *waste*, por lo que la diferencia es casi despreciable, más aún teniendo en cuenta que la política obtenida del método Policy Gradient sacrifica tiempo de respuesta con el fin de no desperdiciar recursos, lo que hace que pase la mitad del tiempo en un estado no recuperable debido al alto tiempo de respuesta promedio.

La política de DQN logra una notable diferencia ante las demás si se tiene en cuenta la permanencia en el estado *optimal*, superando en un 12,02 % a la *baseline*, 43,22 % a la política obtenida del método SARSA y 56,05 % a la de Policy Gradient. Por otra parte, también el método DQN resultó ser el método que logra una política tal que el tiempo de respuesta promedio no sea muy

elevado, es decir, que pasa menos tiempo en los estados *rbp* y *ubp*. En el estado *rbp*, DQN pasó un 10,73 % del tiempo de evaluación, la política *baseline* un 14,92 %, Policy Gradient 18,16 % y SARSA 37,31 %. Por otra parte, DQN no llegó a estar nunca en un estado no recuperable, mientras que la política *baseline* estuvo un 1,98 % del tiempo, SARSA un 12,13 % y Policy Gradient 49,91 %.

Por lo anteriormente presentado, se puede afirmar que el método DQN es el más adecuado entre los estudiados para enfrentar el problema de controlar la elasticidad en servicios de *streaming*. El mismo demostró que al evaluar su política generada a partir del entrenamiento, pasa casi la totalidad del tiempo en un estado óptimo, pudiendo brindar una entrega del servicio a los clientes que no tiene demoras ni interrupciones.

Por otro lado, en las siguientes secciones se presentan los experimentos realizados. En la sección 6.1 se presenta un experimento realizado con el objetivo de verificar si el mal desempeño del método Policy Gradient es debido a la complejidad del sistema o falta de entrenamiento. Mientras que en la sección 6.2, dado que DQN fue el método que presentó resultados satisfactorios a la hora de atacar el problema en estudio, se presentan los distintos experimentos realizados con el objetivo de evaluar en detalle el rendimiento de la política obtenida para dicho método en distintos escenarios.

6.1. Verificación de Policy Gradient

Dado que los resultados obtenidos con este método no fueron satisfactorios, se quiere observar si el método logra resolver el problema de controlar la elasticidad pero en un entorno más simple y predecible, con el objetivo de descartar que el mal desempeño de la política se debiera a falta de tiempo de entrenamiento del agente para adaptarse a las complejidades del problema original. Para ello, se simplificó la complejidad del sistema y se entrenó durante el mismo tiempo que los anteriores pero sobre una única instancia de entrenamiento. En este caso, se utilizó el generador de instancias detallado en la Sección 4.4 del Capítulo 4 para crear un archivo de configuración de manera de utilizarlo en cada episodio del entrenamiento. Esto implica que en cada episodio se tendrá la misma cantidad de clientes en el sistema y cada uno de ellos solicitará el mismo video en el mismo segundo de simulación.

La planificación consiste en un total de 450 clientes que solicitan el video número dos ofrecido por el sistema. Los clientes arriban al sistema de a uno a la vez, desde el segundo cero de la simulación, con una separación de dos segundos entre sí. Al entrenar bajo esta instancia, con la estrategia de *epsilon greedy* y evaluar la política generada de entrenar durante 500 y 1000 episodios, se obtuvieron los resultados que se pueden observar en la Figura 6.2.

Se comparó el resultado del entrenamiento de 500 y 1000 episodios con el objetivo de verificar si a mayor entrenamiento se obtenían mejores resultados.

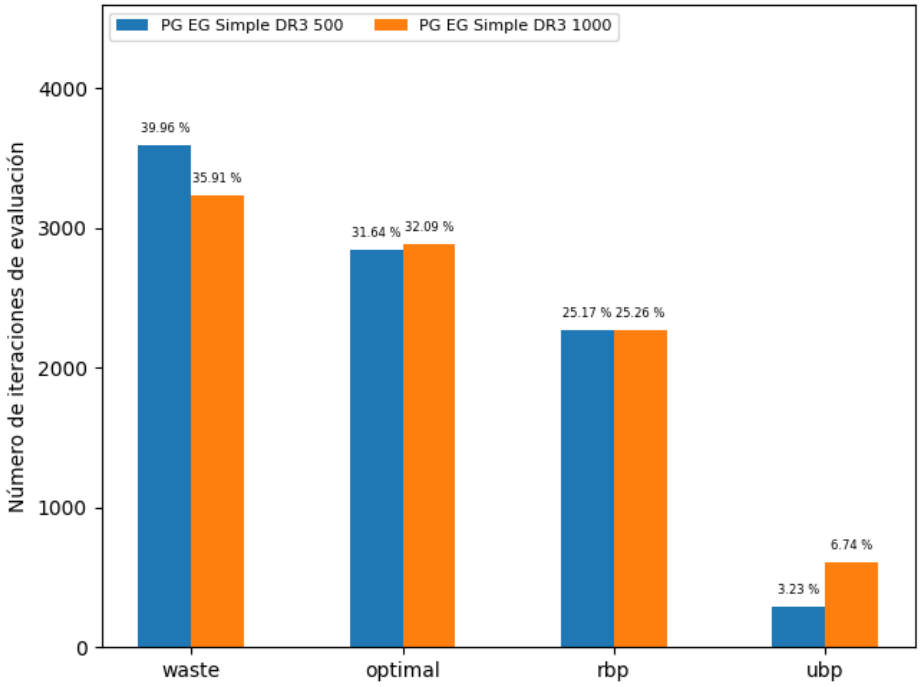

Figura 6.2: Comparación de políticas entrenadas utilizando el método Policy Gradient sobre un espacio de estados discreto con función de recompensa R3, entrenado con *epsilon greedy* sobre una instancia simple, durante 500 y 1000 episodios.

Entrenar 500 episodios más resultó en pasar menos tiempo de la evaluación desperdiciando recursos y más tiempo en un estado óptimo, aunque en este último la diferencia es despreciable. En total, la política entrenada con 1000 episodios, logra estar un 4,05 % menos del tiempo en el estado *waste* que la política entrenada con 500 episodios, mientras que pasa solo un 0,45 % más del tiempo en el estado *optimal*. En contrapartida, al entrenar más se obtuvo una política que genera que el sistema esté más tiempo en el estado no recuperable *ubp*, la política obtenida con 1000 episodios pasó un total de 607 iteraciones en el estado *ubp*, que equivale a 6,74 % del tiempo evaluado, mientras que la obtenida con 500 episodios había estado solo un 3,23 %. Por lo que se puede decir que más entrenamiento no resultó en mejor rendimiento

Por lo tanto, teniendo en cuenta que se entrenaron políticas en entornos de simulación con diferentes características, uno de ellos que representa más fielmente la realidad de un sistema de *streaming* y otro que reduce la complejidad del mismo, y además se evaluaron las políticas obtenidas con distinta cantidad total de episodios entrenados, no es posible concluir que el mal desempeño del

método se deba a una falta de entrenamiento ni a la complejidad del problema original. Se debería realizar otro tipo de experimentación para investigar más a fondo el método y analizar a qué se debe su comportamiento en este problema particular. Esto se plantea como trabajo futuro dadas las limitaciones del proyecto.

6.2. Evaluación de DQN en distintos escenarios

Para validar que la política generada con el método DQN utilizando la función de recompensa R3 se comporta adecuadamente ante distintos escenarios que los de entrenamiento y evaluación, se realizaron un conjunto de experimentos que varían las características del sistema y la capacidad de los servidores que brindan videos a los clientes. Para todos los escenarios evaluados, se utilizó una variante del generador de instancias detallado en la Sección 4.4, con el fin de generar un archivo de configuración que se utiliza para la simulación. De esta manera, cada experimento consiste en una simulación de un episodio de 60 minutos (a diferencia de los anteriores que se realizaron de 15 minutos) y realizando observaciones al entorno cada diez segundos, lo que lleva a tener 360 iteraciones en total.

6.2.1. Experimento 1

El primer experimento busca evaluar si la política es capaz de mantener un buen rendimiento en un episodio de mayor duración, manteniendo las otras características del sistema iguales. De esta manera, se quiere averiguar si lo aprendido en el entrenamiento es escalable en el tiempo y no solo en períodos cortos de simulación.

Los resultados de la evaluación se pueden visualizar en la Figura 6.3 y se verifica el buen funcionamiento de la política bajo este escenario.

En el gráfico a) se observa que en total, de las 360 iteraciones, el sistema estuvo 334 de ellas en el estado *optimal*, 25 iteraciones en *rbp*, solo una iteración en *waste* y ninguna en *ubp*. Esto equivale a que el sistema estuvo un 92,78 % del tiempo en un estado óptimo, 6,94 % del tiempo con el tiempo de respuesta promedio por encima del deseado pero en estado recuperable, y solo 0,27 % desperdiciando recursos. Es importante recalcar que, como en anteriores evaluaciones, el sistema nunca se encontró con un tiempo de respuesta promedio muy elevado, lo cual implica que no debería presentar dificultades para recuperarse.

Por otro lado, en el gráfico b) se visualiza en detalle para cada iteración, en qué estado estuvo el sistema (en color azul), siendo 0 el estado *waste*, 1 el estado *optimal*, 2 el estado *rbp* y 3 el estado *ubp*, y además cuál es el tiempo de respuesta promedio observado (en color naranja). Como se puede observar, las veces que el sistema llegó a estar con un tiempo de respuesta promedio

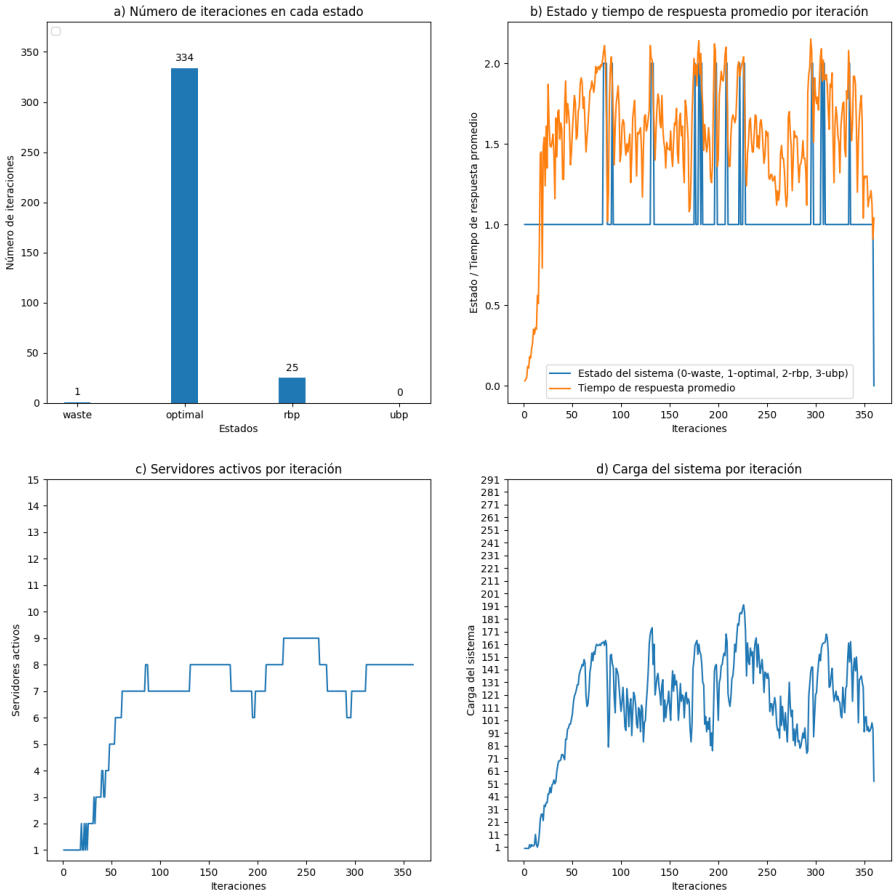

Figura 6.3: Evaluación de la mejor política obtenida de DQN en un episodio de 360 iteraciones.

por encima del deseado (2 segundos), bastaron muy pocas o incluso solo una iteración para volver a un estado óptimo. De hecho, la máxima cantidad de iteraciones que el sistema estuvo por fuera del óptimo, fueron 4 iteraciones seguidas de permanencia en el estado *rbp*, entre las iteraciones 81 y 84. Aún así, pese a esto, en estas iteraciones el sistema estuvo muy cerca del tiempo de respuesta promedio óptimo, no superando los 2.1 segundos. Se nota que al principio de la simulación, el tiempo de respuesta promedio es menor a uno pero igualmente el sistema se encuentra en el estado *optimal*, lo cual significa que, de acuerdo a lo definido en la introducción del Capítulo 5, se tiene la menor cantidad de servidores posibles activos.

Finalmente, si se observan los gráficos c) y d), se puede verificar que las variaciones en la cantidad de servidores activos acompañan a los cambios presentes en la carga del sistema a lo largo de la simulación. También se observa que estos cambios aparecen algunas iteraciones después en el caso de la gráfica c), lo cual puede deberse tanto al tiempo de activación de los servidores, como al tiempo de reacción de la política ante variaciones en la demanda. Un ejemplo de este comportamiento puede observarse en detalle en la iteración 250, en la que la cantidad de servidores se mantiene alta por algunas iteraciones más a pesar de que la carga del sistema se encuentra disminuyendo.

6.2.2. Experimento 2

En este experimento se varió la duración de los tres videos ofrecidos por el sistema. Como se explicó en la Sección 4.4, en el archivo de configuración de un episodio se definen la duraciones en segundos que tienen los videos en la variable *videos_durations*, que por defecto toma 120 segundos para el video 1, 600 segundos para el video 2 y 1200 segundos para el video 3. En este caso, se decidió ofrecer videos de 300, 900 y 1800 segundos respectivamente con el fin de verificar que ante videos más largos, y por tanto, más tiempo de permanencia de los clientes en el sistema, el funcionamiento de la política sigue siendo óptimo.

En la Figura 6.4 se puede observar que se obtuvo un excelente rendimiento en este experimento. En el gráfico a) se puede observar que el sistema pasó casi en su totalidad en un estado óptimo, con muy poco desperdicio de recursos, mientras que nunca superó el tiempo de respuesta promedio deseado. En total, estuvo 355 iteraciones (98,61 %) en el estado *optimal* y 5 iteraciones (1,38 %) en el estado *waste*. Por otro lado, en el gráfico b) se puede observar que el sistema no solo pasó la mayor parte del tiempo con un tiempo de respuesta promedio en el rango deseado, sino que además las pocas ocasiones en las que se encontró en un estado de desperdicio fueron situaciones aisladas de las que pudo recuperarse rápidamente.

En los gráficos c) y d) puede observarse un comportamiento similar al descrito para el Experimento 1, aunque en este caso es interesante resaltar el comportamiento que se da a partir de la iteración 250, en la que si bien la carga del sistema presenta fluctuaciones, la cantidad de servidores se mantiene estable, sin perjudicar el tiempo de respuesta promedio del sistema. Esto podría ser un indicio de que el agente logró anticiparse a las variaciones de demanda, aprendiendo cuando no es conveniente modificar la cantidad de servidores a pesar de observar variaciones en la carga de trabajo.

Finalmente, se puede concluir que la política aprendida no se ajustó a duraciones de videos fijas sino que al variar la duración de los videos, se mantiene el rendimiento deseado.

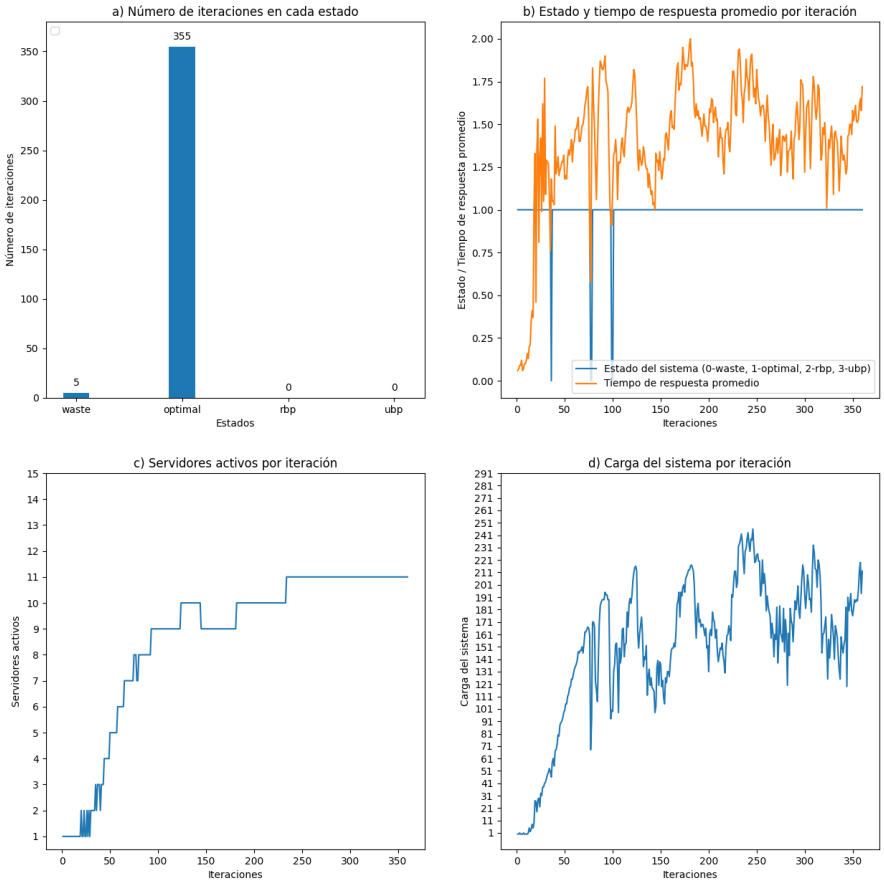

Figura 6.4: Evaluación de la mejor política obtenida de DQN en un episodio de 360 iteraciones con videos de duración 300, 900 y 1800 segundos.

6.2.3. Experimento 3

En este tercer experimento se evalúa el modelo entrenado con cantidades de clientes que presentan grandes variaciones a lo largo de la ejecución. En este caso, se generó una instancia en la que de un segundo al otro, pasa de tener un promedio de cinco nuevos clientes por segundo, a un promedio de 40 a 50, y luego vuelve a cinco. Esto se repite varias veces hasta el final de la instancia. Si bien no es un escenario realista, es útil para ver cómo el modelo se adapta a cambios repentinos en la carga del sistema.

Si bien el rendimiento del modelo en esta instancia es objetivamente inferior al de experimentos anteriores, aún se encuentra dentro de parámetros aceptables.

En la Figura 6.5 se ven los resultados del experimento. Para este escenario, en el gráfico a) se agrega también el comportamiento de la política basada en umbrales para tener una comparación. Se puede observar que la obtenida de entrenar con la función de recompensa R3 estuvo un total de 9 iteraciones (2.5 %) en el estado *waste*, 252 iteraciones (70 %) en el estado *optimal* y 99 iteraciones (27.5 %) en el estado *rbp*, mientras que la basada en umbrales estuvo 41 iteraciones (11,39 %) en *waste*, 270 iteraciones (75 %) en *optimal* y 49 (13,61 %) en *rbp*. Por lo tanto, la política basada en umbrales permanece más tiempo en un estado óptimo y menos en el estado *rbp*, pero desperdicia más tiempo los recursos que la política generada a partir de la función de recompensa R3, que es mínimo.

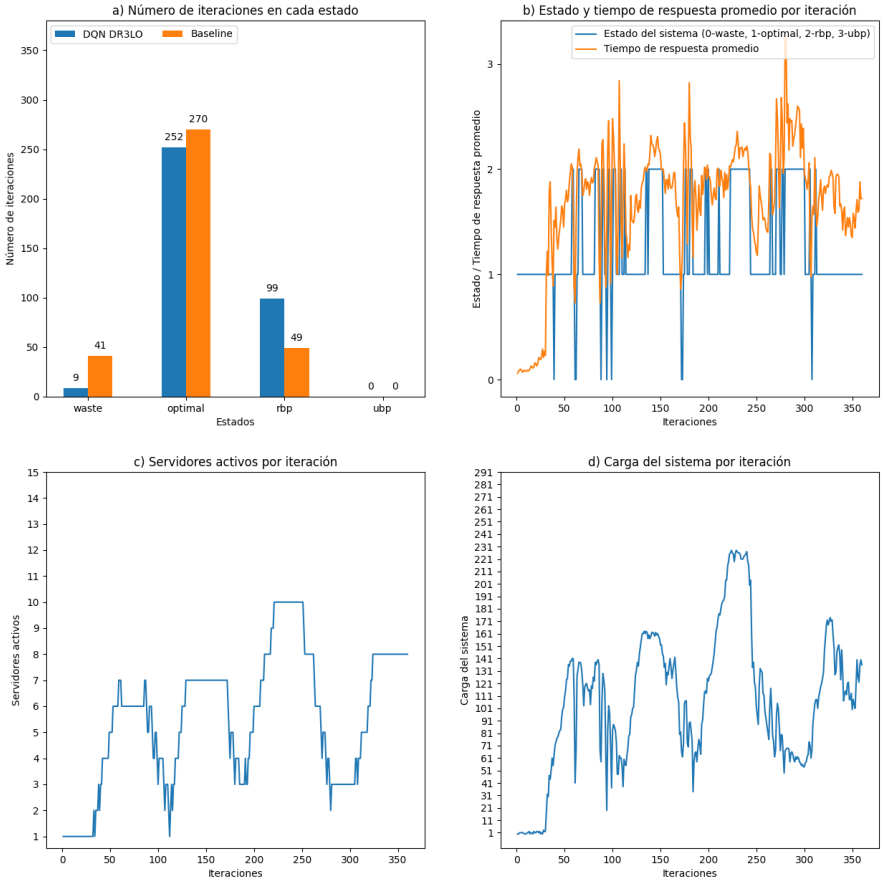

Figura 6.5: Evaluación de la mejor política obtenida de DQN en un episodio de 360 iteraciones con variaciones en cantidades de clientes.

Si bien se observa que permanece más de un cuarto del tiempo en estado *rbp*, al observar los *logs*, en conjunto con el gráfico b), vemos que no suele mantenerse

muchas iteraciones seguidas en ese estado, y cuando lo hace, es con un tiempo de respuesta no muy elevado. Al observar en conjunto con el gráfico d), se puede visualizar que estas situaciones se dan en momentos en los que la cantidad de clientes nuevos por segundo aumenta repentinamente, y eventualmente se vuelve a estabilizar en el estado *optimal*, el cual es un comportamiento esperable ante cambios imprevisibles. Finalmente, al analizar los gráficos c) y d) en conjunto, se puede observar con mayor claridad un comportamiento similar al detallado en los experimentos 1 y 2, en el que la cantidad de servidores activos acompaña a las variaciones en la demanda.

6.2.4. Experimento 4

Para este último experimento se modificaron las características internas del servidor, aumentando al doble el tiempo que tarda en realizar el establecimiento de conexión TCP, en drenar un bloque de archivo y en realizar el *fetch* de un segmento. El objetivo de este experimento es analizar cómo se comporta la política entrenada al ser aplicada en un sistema con características diferentes, de forma de poder evaluar qué consecuencias tuvo haber entrenado con servidores de características fijas sobre la capacidad de adaptación de la política a distintos escenarios.

En la Figura 6.6 se visualizan los resultados del experimento. En el gráfico a) se puede observar que se mantiene en estado *optimal* por 63 iteraciones (17.5 %), en estado *rbp* por 295 iteraciones (81.9 %) y en estado *ubp* por 2 iteraciones (0.5 %). Este comportamiento era previsible ya que el modelo fue entrenado con un servidor con determinadas características, y para esta prueba esas características cambiaron. En el gráfico b) se puede observar que el sistema pasa la mayor parte del tiempo en un estado de mal desempeño considerado recuperable (*rbp*), incluso por momentos estando arriba de 4 segundos, y no consigue quedarse por una cantidad considerable de iteraciones en un estado óptimo. Finalmente, al observar los gráficos c) y d), se puede observar que si bien la variación en la cantidad de servidores activos acompaña a la variación en la demanda, esto no es suficiente para alcanzar el tiempo de respuesta deseado. Esto puede deberse a que la política haya aprendido la cantidad de servidores necesarios para atender correctamente una cierta cantidad de carga teniendo en cuenta las capacidades de los servidores utilizados durante el entrenamiento, y al cambiar las capacidades de cada servidor, dicha cantidad no coincide con la necesaria para el escenario estudiado.

En conclusión, si bien estos resultados no son considerados satisfactorios desde el punto de vista del rendimiento de la política aplicada, son lo suficientemente buenos como para concluir que si se dispusiera del tiempo necesario para entrenar el modelo incluyendo instancias con servidores de diversas características, se obtendrían buenos resultados.

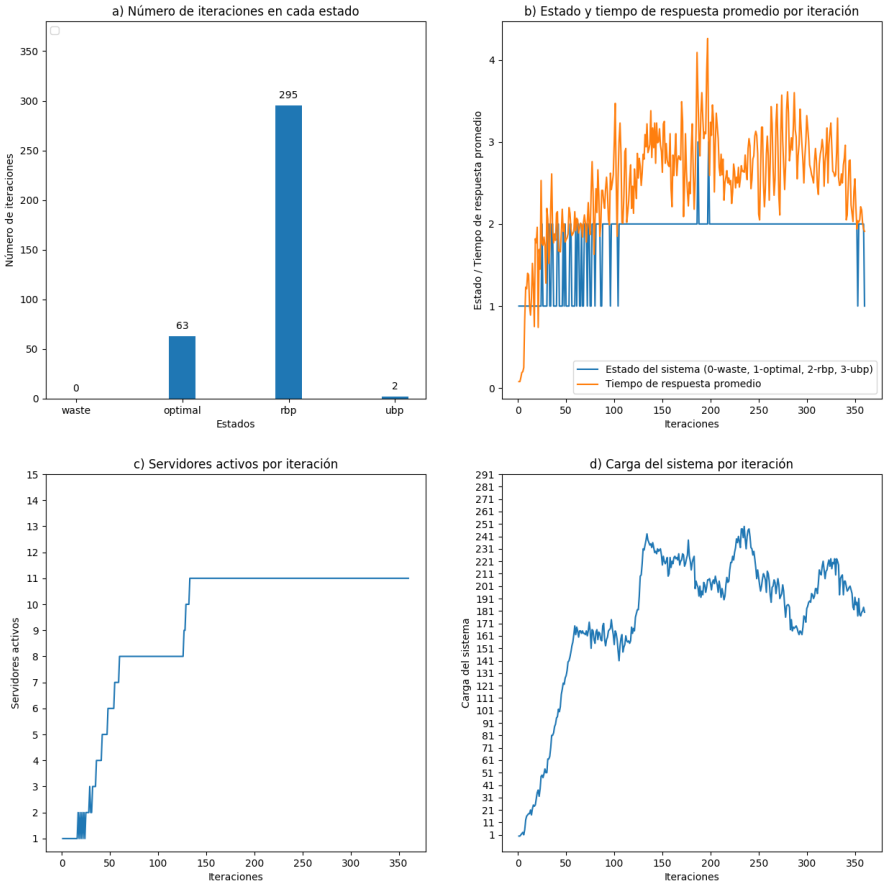

Figura 6.6: Evaluación de la mejor política obtenida de DQN en un episodio de 360 iteraciones en un sistema HAS con servidores de menor capacidad.

Capítulo 7

Conclusiones y trabajo futuro

Como resultado del trabajo realizado en el marco del Proyecto de Grado, se logró implementar un entorno de trabajo que permite entrenar, evaluar y aplicar estrategias de Aprendizaje por Refuerzo para controlar la elasticidad de un sistema simulado de *streaming* de video adaptativo basado en HTTP. El entorno de trabajo incluye:

- un simulador diseñado específicamente para este fin, con características configurables, que permite representar sistemas diversos. El simulador cuenta con un conjunto de servidores que ofrecen videos, una representación de los clientes que solicitan segmentos de estos videos y un balanceador de carga que se encarga de recibir estas solicitudes y decidir cuál servidor la atenderá de acuerdo a un criterio establecido.
- un controlador de elasticidad que recibe como entrada una política e interactúa con el simulador, ejecutando acciones para llevar a cabo un control de elasticidad horizontal, pudiendo agregar o quitar un servidor de acuerdo a observaciones que hace sobre el sistema, teniendo en cuenta las métricas observadas.
- un módulo de entrenamiento que permite entrenar agentes de Aprendizaje por Refuerzo utilizando distintos métodos. Este módulo se lleva a cabo mediante la interacción de un agente con el entorno, que le brinda recompensas por las buenas acciones y lo penaliza por las malas acciones, logrando de esta manera aprender una política que es capaz de decidir qué acción tomar de acuerdo al estado del sistema.
- un módulo de evaluación que permite medir el rendimiento de las políticas generadas a partir del entrenamiento. Para evaluar una política se definieron cuatro estados que indican un cierto nivel de rendimiento del

sistema: *waste*, *optimal*, *recoverable bad performance* y *unrecoverable bad performance*. De esta manera, observando el tiempo de permanencia del sistema en cada estado se puede medir qué tan buena es una política.

- una política *baseline* contra la que comparar las distintas políticas obtenidas del entrenamiento. La misma sigue una lógica simple de acción que consta de subir o quitar servidores de acuerdo al tiempo de respuesta promedio observado.

Utilizando este sistema, se logró implementar, entrenar y evaluar tres métodos de Aprendizaje Automático, proceso que fue descrito en los anteriores capítulos. De acuerdo a los resultados observados, se llegó a las siguientes conclusiones: dado un sistema de *streaming* con las características que se determinaron, los modelos basados en los métodos de Policy Gradient y SARSA no dan resultados satisfactorios habiendo sido entrenados por 1000 episodios. Sin embargo, el modelo entrenado utilizando el método DQN, en tan solo 1000 episodios logró un rendimiento superior al de una heurística basada en umbrales. Por lo tanto, se concluye que este modelo es capaz de asignar recursos de manera eficiente, pudiendo mantener en un rango óptimo los tiempos de respuesta hacia los clientes, logrando este objetivo, además, evitando el desperdicio de recursos. Además, es importante recordar que este resultado se pudo obtener luego de discretizar el espacio de estados posibles, cambio que redujo severamente la cantidad de estados con los que entrenar el modelo, permitiéndole al modelo explorar más estados en menos tiempo y facilitando el entrenamiento.

Es de interés observar que estos resultados dependen de varios aspectos: la naturaleza del problema, la construcción del simulador y las características de los métodos utilizados. El método de DQN, si bien funciona de manera similar al método SARSA, cuenta con una red neuronal, que le da una ventaja a la hora de enfrentarse al problema y representar su complejidad. Esta ventaja se debe a la capacidad de las redes neuronales de adaptarse a los datos de entrenamiento y reconocer patrones sutiles para realizar predicciones. Por otro lado, el método Policy Gradient también cuenta con una red neuronal, pero su capacidad de aprendizaje se ve restringida por la tendencia que sufre a quedarse en un óptimo local, por más que fuera forzado a explorar nuevos estados. Se entiende que es por estos motivos que entrenar un modelo con DQN da mejores resultados para este tipo de problemas complejos, con un espacio de estados muy amplio.

Durante la realización de este proyecto surgieron diversas dificultades. En primer lugar, se presentó el desafío de enfrentarse a conceptos y herramientas desconocidas, esto se debe a que el equipo nunca había trabajado con simulaciones ni tampoco con Aprendizaje por Refuerzo, lo que requirió que se dedicara un tiempo considerable al comienzo del proyecto a investigar y familiarizarse con el tema de estudio y todas sus componentes.

Por otro lado, los tiempos de entrenamiento elevados y la escasez de recursos dificultaron la velocidad de avance durante la realización del proyecto, ya que en muchas ocasiones era necesario esperar entre dos a tres semanas de entrena-

miento antes de poder evaluar los resultados de cualquier cambio, para recién entonces poder definir cómo proceder a continuación. A su vez, al ser un proyecto que involucra estudio y exploración, muchos de esos cambios y tiempo invertido no se vieron reflejados en el resultado final, ya que no todos llevaron a resultados favorables o fueron pasos en la dirección correcta.

Sin embargo, a pesar de las dificultades encontradas, se logró cumplir satisfactoriamente con los objetivos principales del proyecto.

Como trabajo a futuro, tomando de base el sistema construido, se podría realizar una búsqueda exhaustiva de hiperparámetros, para encontrar la mejor combinación que ayude a la hora de entrenar un modelo. Realizar esto requeriría gran cantidad de tiempo de entrenamiento, pero podría permitir obtener mejores resultados. Otra mejora a realizar, que también ayudaría en la búsqueda de hiperparámetros, es la optimización del simulador. Por ejemplo, se podrían simplificar algunas características del mismo, sacrificando fidelidad con la realidad por un tiempo de simulación más eficiente, lo cual resultaría en un menor tiempo de entrenamiento.

Otra mejora posible, que sería facilitada por tener menor tiempo de entrenamiento, sería la experimentación con distintos métodos. También se podrían utilizar los modelos obtenidos para realizar aprendizaje por transferencia en un entorno real. Es decir, tomar como base lo entrenado, y continuar con el entrenamiento de ese modelo utilizando un sistema real en vez de una realidad simulada.

Referencias

Arulkumaran, K., Deisenroth, M. P., Brundage, M., y Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38. doi: 10.1109/MSP.2017.2743240

Bitsakos, C., Konstantinou, I., y Koziris, N. (2018). Derp: A deep reinforcement learning cloud system for elastic resource provisioning. En *2018 ieee international conference on cloud computing technology and science (cloudcom)* (p. 21-29). doi: 10.1109/CloudCom2018.2018.00020

Brunton, S. L., y Kutz, J. N. (2019). *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press. doi: 10.1017/9781108380690

Dick, T. B. (2015). Policy gradient reinforcement learning without regret.

Gerón, A. (2019). *Hands-on machine learning with scikit-learn, keras, and tensorflow: Concepts, tools, and techniques to build intelligent systems* (Second ed.). O'Reilly.

González, S., Castellanos, W., Guzmán, P., Arce, P., y Guerri, J. C. (2016). Simulation and experimental testbed for adaptive video streaming in ad hoc networks. *Ad Hoc Networks*, *52*, 89-105.

Larraura, M., Rotti, M., y Tito Virgilio, S. (2022). *Simulador de sistema de streaming adaptativo basado en http.* https://eva.fing.edu.uy/pluginfile.php/324214/mod_folder/content/0/InformeTSCF_Simulador_Streaming.pdf?forcedownload=1.

Lolos, K., Konstantinou, I., Kantere, V., y Koziris, N. (2017). Elastic management of cloud applications using adaptive reinforcement learning. En *2017 ieee international conference on big data (big data)* (p. 203-212). doi: 10.1109/BigData.2017.8257928

MathWorks. (s.f.). *Call matlab from python.* https://la.mathworks.com/help/matlab/matlab-engine-for-python.html. (Accessed: 2023-06-27)

MathWorks. (2016). *Matlab discrete-event system.* https:/la.mathworks.com/help/simevents/ref/matlabdiscreteeventsystem.html?s_tid=srchtitle_discrete-event%20system_1. (Accessed: 2023-06-20)

Mitchell, T. M. (1997). *Machine learning* (E. M. Munson, Ed.). Boston, MA: WCB/McGraw-Hill.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015, febrero). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Ott, H., Miller, K., y Wolisz, A. (2017 June). Simulation framework for http-based adaptive streaming applications. *Proceedings of the 2017 Workshop on ns-3*, pp. 95-102.

PyLessons. (s.f.). *Epsilon greedy in deep q learning.* https://pylessons.com/Epsilon-Greedy-DQN#.

Subbiah Natarajan. (s.f.). *Stretched exponential decay function for epsilon greedy algorithm.* https://medium.com/analytics-vidhya/stretched-exponential-decay-function-for-epsilon-greedy-algorithm-98da6224c22f.

Sutton, R. S., y Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second ed.). The MIT Press.

Tsoumakos, D., Konstantinou, I., Boumpouka, C., Sioutas, S., y Koziris, N. (2013). Automated, elastic resource provisioning for nosql clusters using tiramola. En *2013 13th ieee/acm international symposium on cluster, cloud, and grid computing* (p. 34-41). doi: 10.1109/CCGrid.2013.45

Van der Mei, R. D., Hariharan, R., y Reeser, P. K. (2001). Web server performance modeling. *Telecommunication Systems*, *16*, 361–378.