

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“OPTIMIZACIÓN DE *PERFORMANCE* PARA CONSULTAS SPARQL UTILIZANDO *DEEP LEARNING*”

CRISTÓBAL IGNACIO CARMONA RODRIGUEZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Buil Aranda
Profesor Correferente: Daniel Casals

Enero 2022

DEDICATORIA

Le dedico este trabajo a mis amigos más cercanos, pero sobre todo, a mi familia que siempre brindaron un apoyo incondicional durante todo este proceso, y siempre me tuvieron fe. Sin ellos, hubiera sido prácticamente imposible culminar este extenso proceso.

AGRADECIMIENTOS

Las primeras personas con quienes estaré eternamente agradecido son a mis padres, Carlos Carmona Acosta y Guacolda Rodríguez Gana, por su inagotable apoyo práctico, económico y por sobre todo emocional. Gracias por haberme permitido y por haberme ayudado a embarcar esta extraña aventura lejos de casa, que ya está próxima a terminar. Quisiera agradecer también a mi tío Armando "quico" Rodríguez por recibirme, alojarme y ayudarme en los siempre difíciles primeros años de la carrera. También agradecer a mi hermano Carlos Carmona Rodríguez por su apoyo emocional y por hacerme compañía durante algunos años. Por supuesto también le agradezco al resto de mi familia por su apoyo, principalmente a mi abuelita Ismenia Gana, ya que tuve la suerte de pasar toda la cuarentena con ella, coincidiendo con el tiempo de realización de la memoria, y que me permitió conocerla mucho más, y siempre estuvo dispuesta a ayudarme como ella pudiera, ya sea cocinando, conversando, etc.

No quisiera olvidar a los amigos que hice durante el transcurso de la carrera, principalmente a mis amigos Obriel Muga y Eduardo Reyes, con los cuales compartí muchas experiencias, alegrías y conocimientos, así como también nos apoyamos mutuamente en los momentos complicados y difíciles, pasando penas y rabias similares. Sin estós dos hubiera sido muy distinto todo el camino recorrido. También quisiera agradecer a mis amigos de toda la vida, con los cuales me desconecto de la rutina, los cuales fueron un apoyo emocional, y que me dieron pequeños pero importantes momentos de alegría en situaciones complejas, de estrés, de pena y de incertidumbre, la mayoría de las veces sin saberlo.

Quisiera agradecer a los profesores de la UTFSM que me aportaron los conocimientos y herramientas para poder terminar este proceso. Muy especialmente, agradezco al profesor guía Carlos Buil-Aranda por darme la oportunidad de realizar esta memoria, por tenerme paciencia, por guiarme y ayudarme en los momentos en que se complicó el trabajo. Finalmente, quisiera agradecer al exalumno Daniel Casals Amat, el cuál siempre estuvo dispuesto a resolver mis dudas. Gracias a ambos, ya que hicieron mucho más ameno y satisfactorio la realización de esta memoria.

RESUMEN

Resumen— La optimización de *performance* en las consultas, es uno de los desafíos más importantes en los gestores de base de datos, y no está tan desarrollado en las bases de datos de grafo que usan SPARQL como lenguaje de consulta. Se propone enfrentar el problema mediante modelos de *Deep Learning*, utilizando en particular, la variante *Deep Reinforcement Learning*. Para poder desarrollar estos modelos, fue necesario enfrentar varios desafíos relacionados: extracción y caracterización de las consultas, la implementación de un modelo *Deep Learning* para predecir latencia, reeplantear el problema como un proceso de decisión de Markov, construir una base para modelos de *Deep Reinforcement Learning*, y la implementación de algoritmos de *Reinforcement Learning*. Los modelos fueron capaces de entregar, en promedio, un mejor plan de consulta que los propuestos por el gestor de base de datos *Virtuoso Open Source*. Como pocas veces se ha abordado este problema en SPARQL usando este tipo de modelos, se abren buenas oportunidades para trabajos futuros.

Palabras Clave— Optimización, latencia, consulta, SPARQL, *Deep Reinforcement Learning*

ABSTRACT

Abstract— Query *performance* optimization is one of the most important challenges in database managers, and it is not so well developed in network databases that using SPARQL as query language. It is proposed to face the problem by means of *Deep Learning* models, using, in particular, the *Deep Reinforcement Learning* variant. In order to develop these models, it was necessary to face several related challenges: extraction and characterization of the queries, implementation of a *Deep Learning* model to predict latency, restate the problem as a Markov decision process, build a basis for *Deep Reinforcement Learning* models, and the implementation of *Reinforcement Learning* algorithms. The models were able to deliver, on average, a better query plan than those proposed by the *Virtuoso Open Source* database manager. As this problem has rarely been addressed in SPARQL using this type of models, it opens up good opportunities for future work.

Keywords— Optimization, latency, query, SPARQL, *Deep Reinforcement Learning*

GLOSARIO

BD: Base de datos

RDB: Base de datos relacionales, en inglés *Relational Database*

RDF: Marco de Descripción de Recursos, en inglés *Resource Description Framework*

SPARQL: Lenguaje de consultas estandarizado para grafos RDF

BGP: Patrón de grafo básico, en inglés *Basic Graph Pattern*

SCAN: En consultas SPARQL, una consulta SCAN se refiere a la ejecución parcial sobre un predicado en un triple.

W3C: *World Wide Web Consortium*

VOS: *Virtuoso Open Source*

Latencia: Tiempo de ejecución de una consulta

MDP: Proceso de decisión de Markov, en inglés *Markov Decision Process*

ML: Aprendizaje de maquinas, en inglés *Machine Learning*

DL: Aprendizaje profundo, en inglés *Deep Learning*

Neo: Hace referencia al proyecto Neo: *A Learned Query Opimize* [Marcus et al., 2019]

TreeConv: Convolución de árboles, en inglés *Tree Convolution*

MSE: Error cuadrático medio, en inglés *Mean Squared Error*

RMSE: Raíz del error cuadrático medio, en inglés *Root Mean Squared Error*

MAE: Error absoluto medio, en inglés *Mean Absolute Error*

RL: Aprendizaje por refuerzo, en inglés *Reinforcement Learning*

DRL: Aprendizaje por refuerzo profundo, en inglés *Deep Reinforcement Learning*

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Contexto	2
1.2 Objetivos	4
1.2.1 Objetivo General	4
1.2.2 Objetivos específicos	4
CAPÍTULO 2: MARCO CONCEPTUAL	5
2.1 Conceptos técnicos	5
2.1.1 Optimizador de consultas	5
2.1.2 <i>Resource Description Framework</i> - RDF	7
2.1.3 SPARQL	9
2.1.4 <i>Machine Learning</i> y <i>Deep Learning</i>	11
2.1.5 Proceso de decisión de Markov	15
2.1.6 <i>Reinforcement Learning</i>	15
2.1.7 Elementos de RL	16
2.1.8 Flujo general de un problema RL	17
2.2 Herramientas	19
2.2.1 <i>Virtuoso Open Source</i>	19
2.2.2 Otras herramientas computacionales	21
2.2.3 Hardware	21
2.3 Estado del arte	21
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	26
3.1 Extracción y caracterización de los datos	26
3.1.1 Extracción de los datos	26
3.1.2 Caracterización a nivel de consulta	29
3.1.3 Caracterización a nivel de plan de ejecución	29
3.1.4 <i>Dataset</i>	31
3.2 Diseño de modelo DRL	33
3.2.1 Deep Learning	33
3.2.2 Reinforcement Learning	37

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	46
4.1 Predicción de latencia	46
4.1.1 Métricas de Evaluación	46
4.1.2 <i>Tuning</i> de hiperparámetros	47
4.1.3 Análisis al mejor modelo de predicción de latencia obtenido	53
4.2 Optimización de consultas	56
4.2.1 <i>Tuning</i> de hiperparámetros	56
4.2.2 Análisis a los mejores modelos DRL obtenidos	64
CAPÍTULO 5: CONCLUSIONES	69
5.1 Conclusiones Generales	69
5.2 Conclusiones de resultados	70
5.2.1 Resultado del modelo de predicción de latencia desarrollado	70
5.2.2 Resultados de los modelos de optimización de consulta desarrollados . .	71
5.3 Oportunidades de mejora	73
5.4 Trabajos futuros	74
ANEXOS	75
5.1 Pseudocódigos para distintos algoritmos de RL	75
REFERENCIAS BIBLIOGRÁFICAS	78

ÍNDICE DE FIGURAS

1	Arquitectura de un optimizador de consultas con enfoque basado en costo. . .	6
2	Un RDF-triple con dos nodos (Bob y Alice) y un IRI que los conecta (<i>is a friend of</i>). En este caso, Bob es el sujeto, <i>is a friend of</i> es el predicado y Alice es el objeto.	8
3	Grafo RDF generados por los triples del ejemplo.	9
4	Ejemplo de consulta SPARQL.	11
5	Comparación entre Ingeniería de Software y Machine Learning.	12
6	Ejemplo de MDP.	16
7	Diagrama de RL donde se refleja la interacción entre Agente y Entorno.	18
8	Pseudocódigo de algoritmo <i>Q-Learning</i> , con algunos comentarios que indican que se está haciendo.	18
9	Ejemplo de traducción de una consulta SPARQL a una consulta SQL en VOS. La consulta traducida es la presentada en la Figura 4.	20
10	Diagrama de la <i>Neo's value network</i>	23
11	Diagrama de una política en Neo, donde se puede apreciar el valor en la política <i>Q</i> de cada estado. En rojo se remarca el mejor árbol, y por ende, plan de ejecución, posible.	24
12	Estructura en formato árbol de una consulta SPARQL.	25
13	Ejemplo de sentencia PROFILE ejecutado por VOS en la consulta de la Figura 4.	28
14	Histograma de latencia de las consultas sin procesar, hasta un máximo de 200 milisegundos de latencia.	32
15	Ejemplo de <i>Tree Convolution</i>	37
16	Pseudocódigo para optimizar una consulta SPARQL.	40
17	Ejemplo de flujo del proceso de generar un nuevo árbol de ejecución de consultas en una consulta genérica con dos BGPs (uno de ellos del tipo OPTIONAL), cuatro subconsultas, y tres predicados. Este proceso se realiza en cada iteración.	42

18	Ejemplo de como varía una política utilizando el algoritmo Neo, junto con el árbol final que generaría para una consulta genérica con tres BGPs (uno de ellos del tipo OPTIONAL), cinco subconsultas, y cuatro predicados.	43
19	RMSE y MAE por iteración, en modelos similares con distintos optimizadores. .	48
20	RMSE y MAE por iteración, en modelos similares con distintas tasas de aprendizaje.	50
21	RMSE y MAE por iteración, en modelos similares con distintos tamaños de capas ocultas.	52
22	Comparación de valores de latencia reales versus las predicciones del modelo final.	54
23	RMSE y MAE del modelo final.	55
24	Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de ε	57
25	Convergencia de cada algoritmo con distintos valores de ε	58
26	Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de α	60
27	Convergencia de cada algoritmo con distintos valores de α	61
28	Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de γ	62
29	Convergencia de cada algoritmo con distintos valores de γ	63
30	Promedio de latencia de los mejores modelos DLR, junto con la convergencia de cada uno de estos.	64
31	Promedio de latencia de los mejores modelos DLR, separados por numero de JOINS.	65
32	Errores totales de cada modelo DLR.	66
33	Errores de cada modelo DLR separados en distintos rangos de latencia.	67
34	Errores de cada modelo DLR separados por número de JOINS.	68

INTRODUCCIÓN

El problema de optimización de las consultas es uno de los desafíos más grandes a los que se enfrentan los gestores de base de datos. En palabras simples, un motor de base de datos al momento de ejecutar una consulta genera internamente un plan de consultas, el cual describe básicamente como se ejecutó esta consulta: el orden de las subconsultas, la cardinalidad, tablas índices que se utilizarán, entre otros. La comunidad de base de datos relacionales a hecho grandes avances al respecto, incluyendo modelos de costo, heurísticas, estimadores de cardinalidad, y modelos utilizando técnicas de *Machine Learning*. Por el contrario, en las bases de datos de grafo RDF, que ocupan SPARQL como lenguaje de consulta, aún existen varias oportunidades de mejoras al respecto. Existen trabajos en SPARQL que han logrado predecir la latencia de las consultas, pero pocos de estos se enfocan en optimizar las consultas propiamente tal.

En esta memoria se realizaron propuestas de modelos de *Deep Learning*, en particular, utilizando el paradigma *Deep Reinforcement Learning*, los cuales permiten encontrar mejores planes de ejecución de consultas que un gestor de base de datos RDF común como lo es *Virtuoso Open Source*. Para poder realizar estos modelos, hubo que enfrentar tres problemas fundamentales: la extracción y caracterización de las consultas, la implementación de un modelo supervisado de *Deep Learning* para la predicción de latencia, y el desarrollo e implementación de modelos de *Reinforcement Learning*, que usa el modelo de predicción para optimizar las latencias de las consultas, siempre respetando las propiedades del lenguaje SPARQL. En el Capítulo 1 se da un contexto general al problema, explicando concisamente los conceptos más importantes relacionados con el trabajo realizado. Además, se hace un repaso breve en el estado del arte a los trabajos más influyentes. En el Capítulo 2, se detalla la solución propuesta, explicando el funcionamiento del extractor de características, las características extraídas, la elección del *dataset*, el proceso de implementación del modelo junto al funcionamiento de este, el replanteo del problema como un MDP, y el diseño e implementación de los modelos de RL. En el Capítulo 3 se explica como se evaluaron los modelos, se hace un *tuning* de hiperparámetros¹ y se hace un análisis comparativo entre los modelos y los valores reales. Finalmente en el Capítulo 4, se concluye sintetizando el trabajo realizado, los resultados obtenidos, las oportunidades de mejora a la memoria propiamente tal, y los trabajos futuros que pueden realizarse a partir de esta.

¹El *tuning* de hiperparámetros, es el proceso de afinamiento y configuración de parámetros ajustables que mejoran el desempeño de una aplicación o modelo.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. Contexto

Como se mencionó en la introducción, la optimización de consultas, o *query optimization*, es uno de los desafíos más grandes que deben enfrentar los gestores de bases de datos. Este tiene como objetivo principal, mejorar el rendimiento, también llamado *performance*, de las consultas o *queries*, de tal manera que el tiempo de ejecución sea óptimo. En otras palabras, **se busca que la latencia de las consultas tengan el menor valor posible**. Por lo tanto, es común que los gestores de bases de datos, tengan optimizadores de consultas internos, los cuales pueden acelerar la ejecución de consultas en varios órdenes de magnitud. Estos optimizadores básicamente transforman una consulta del lenguaje del gestor de base de datos que se está utilizando (SQL, SPARQL, etc.) a un **plan de consultas**, lo cual será lo que se ejecutará realmente. Sin embargo, construir un buen optimizador hoy en día, toma miles de horas de ingeniería, inteligencia y esfuerzo, necesitando de bastante personal, y es una disciplina que solo unos pocos expertos dominan por completo. Peor aún, los optimizadores de consultas deben tener una tediosa mantención, especialmente a medida que evolucionan los motores de ejecución y la capacidad de almacenamiento de los sistemas. Esto provoca que ninguno de los optimizadores de consultas de código abierto disponibles de forma gratuita se acerque al *performance* de los optimizadores comerciales ofrecidos por IBM, Oracle o Microsoft [Marcus *et al.*, 2019].

Por otra parte, en los últimos años se ha popularizado el uso de Bases de Datos no Relacionales y/o Distribuidas (también llamadas NoSQL), en respuesta a las limitantes que tienen las Bases de Datos Relacionales (también llamadas RDB o SQL) [scalegrid.io, 2019]. Algunas de las ventajas más importantes que ofrecen las Bases de Datos NoSQL son que crecen horizontalmente mejor, tienen mayor flexibilidad y usualmente funcionan mejor con grandes cantidades de datos en comparación a sus contrapartes relacionales. Las Bases de Datos Orientadas a Grafos, o GDB, son un tipo de Base de Datos NoSQL, y un modelo bastante utilizado son las **Resource Description Framework (o bases de datos RDF)**, el cual es el estándar recomendado por el consorcio W3C para representar información sobre recursos en la *World Wide Web*, y su uso es clave en la Web Semántica [W3C *et al.*, 2014a]. **SPARQL es el lenguaje de consultas estándar utilizado por RDF** y a sido implementado por varios motores [ApacheJena, 2020][OpenLinkSoftware, 2021][Erling y Mikhailov, 2009] que admiten cientos de *endpoints* públicos en la Web que reciben miles de consultas diarias [Buil-Aranda *et al.*, 2013]. También ha sido utilizado en diversos proyectos de la Web Semántica, como lo es *Linked Open Data* [ontotext, nd]. Las implementaciones de SPARQL funcionan bastante bien para consultas simples, incluso si la carga de trabajo es alta. No obstante, cuando se trabaja con consultas complejas, o varias consultas a la vez, se generan

conflictos al evaluarlas, provocando como es de esperarse, mayores tiempos de espera y un mayor gasto de recursos en procesar estas consultas. Algunas soluciones que se han planteado para resolver este problema consisten en que el optimizador de consultas interno utilice estadística de datos y heurísticas como base [Tsialiamanis *et al.*, 2012]. Sin embargo, tienen que hacer suposiciones fuertes, como considerar las consultas de cierta estructura menos costosas que otras. Estas suposiciones pueden ser válidas para algunos conjuntos de datos RDF, pero para otros no, lo que termina haciendo que estos optimizadores sean muy irregulares, ya que las bases de datos RDF no tienen un esquema claro a diferencia de las SQL [Hasan y Gandon, 2014].

La optimización de consultas ha tenido varios avances si hablamos de bases de datos SQL, en donde se han utilizado una gran variedad de técnicas, que incluye el uso de algoritmos de *Machine Learning* (ML), incluido algoritmos de *Deep Learning* (DL), un subcampo muy importante del primero. Cuando se aplican este tipo de algoritmos, se deben tomar en cuenta varios puntos relacionados, como por ejemplo: la calidad y cantidad de datos del *dataset* para los conjuntos de entrenamiento, validación y prueba, como se extraerán estos datos, qué características de los datos se seleccionarán, qué modelo se utilizará (regresión, clasificación, entre otros), como se *tunearan* los hiperparámetros, etc. Un ejemplo es Neo[Marcus *et al.*, 2019] donde se utilizó **Deep Reinforcement Learning** (DLR), el cual es una variante del DL que utiliza los principios del *Reinforcement Learning* (RL), con un punto de vista *end-to-end*. Además, realizaron una caracterización basada en árboles y un modelo de predicción de latencia que usa una arquitectura de redes neuronales que desarrollaron y denominaron *Value Network*, donde la principal característica es que usa un tipo red neuronal de convolución de árboles o *Tree Convolution*. Neo puede predecir el rendimiento de la ejecución de consultas o subconsultas bajo ciertos supuestos bastante bien, obteniendo mejores resultados que los optimizadores de consultas oficiales, como el de Microsoft SQL-Server y el de Oracle. Sin embargo, estas técnicas no han sido tan exploradas en las bases de datos RDF (en comparación a las bases de datos relacionales), por lo que existen grandes oportunidades de mejora, ya sea en la caracterización de los datos, en los modelos a utilizar, la cantidad de datos, etc. También podrían aprovecharse los trabajos hechos en bases de datos relacionales e implementar, adaptar o desarrollar ideas inspiradas de estos trabajos. De todas maneras, existen algunos proyectos interesantes como [Hasan y Gandon, 2014], en donde se utilizaron modelos de ML basados en regresión y dos caracterizaciones de las consultas utilizando propiedades de las bases de datos RDF. No obstante, también se cometieron algunos errores (como la poca cantidad de datos de entrenamiento) los cuales mermaron los resultados obtenidos. Otro trabajo relacionado es el realizado en la tesis de magíster de Daniel Casals [Casals *et al.*, 2020], ex-estudiante de postgrado de la Universidad Técnica Federico Santa María, el cual es similar al proyecto mencionado anteriormente, pero con la gran diferencia de que se utilizó un mejor *dataset*, con una mayor cantidad de datos de entrenamiento, además de implementar un modelo de DL que emplean las *Tree Convolution* de Neo mencionados anteriormente. Ambos proyectos tuvieron como enfoque la predicción de latencias de consultas SPARQL, más no la optimización de estas.

1.2. Objetivos

1.2.1. Objetivo General

Investigar y aplicar técnicas de *Deep Learning* para mejorar el *performance* de consultas SPARQL, mejorando los tiempos de respuesta en una Base de Datos Orientada a Grafo RDF.

1.2.2. Objetivos específicos

- Explorar nuevas formas de representar los datos de consultas SPARQL.
- Proponer y diseñar modelos de *Deep Learning* orientados a mejorar los *performance* de consultas SPARQL.
- Comparar modelos de *Deep Reinforcement Learning* y analizar cuál es mejor.

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Conceptos técnicos

En esta sección se explicarán con mayores detalles ciertos conceptos técnicos de manera breve, necesarios para entender el resto de la memoria.

2.1.1. Optimizador de consultas

El objetivo principal de la optimización de consultas es bajar la latencia al ejecutar consultas, es decir, reducir el tiempo de ejecución de estas. Para esto los sistemas de bases de datos utilizan optimizadores de consultas internos, en donde el optimizador busca generar y ejecutar el plan de consultas más óptimo de una consulta dada. Los optimizadores de consultas son clave para lograr un buen desempeño en los sistemas de bases de datos, y pueden acelerar la ejecución de consultas en varios ordenes de magnitud. No obstante, construir un buen optimizador hoy en día es complicado, ya que requiere un gran esfuerzo humano y material, y pocos dominan a nivel experto esta disciplina [Marcus *et al.*, 2019]. La razón de esto, es que un optimizador de consultas contiene muchos componentes que interactúan entre sí. La arquitectura clásica de un optimizador es la que tiene un enfoque basado en costo [Leis *et al.*, 2015] [Medina, 2010]. En la Figura 1 se puede ver un diagrama sencillo de como una consulta es procesada por un optimizador para construir el plan de consultas. A continuación se detallarán brevemente los componentes más importantes:

- **Estimación de cardinalidad**

La estimación de cardinalidad es el componente más importante para encontrar un buen plan de consulta. El objetivo de la estimación de la cardinalidad es, como dice su nombre, estimar el tamaño de las partes intermedias de una consulta. Por ejemplo, si tengo una consulta con varios JOIN's, se usaría la estimación de cardinalidad para descubrir cuáles expresiones se unen para hacer el primer JOIN, cuales para el segundo, cuales para el tercero, y así sucesivamente, en función del tamaño. Muchos trabajos suelen concentrarse en mejorar la estimación de la cardinalidad con la esperanza de mejorar la latencia de las consultas. No obstante, una buena estimación de cardinalidad no necesariamente podría mejorar un plan de consultas, ya que no existe una correlación directa entre ambos. En realidad, las estimaciones de cardinalidad generalmente se calculan en función de supuestos simplificadores como la uniformidad y la independencia. En los *datasets* del mundo real, estas suposiciones a menudo son incorrectas, lo que puede conducir a planes subóptimos. Para la estimación de cardinalidad se utilizan varios métodos destacando principalmente el uso de histogra-

mas, pero también hay algunos más avanzados como el uso de listas de valores más frecuentes, empleado en el estimador de PostgreSQL [Leis et al., 2015].

- **Modelo de costo**

El objetivo del modelo de costos es tomar la información dada por la estimación de la cardinalidad para obtener el costo de cada plan de consultas posible. Estos modelos suelen ser funciones aritméticas polinomiales.

- **Plan space enumeration algorithm**

Este componente explora el espacio consultas semánticamente equivalentes en el plan de consultas generado utilizando la estimación de cardinalidad y el modelo de costo. Si el espacio de los planes de consultas es muy grande, se utilizarán soluciones como algoritmos heurísticos o soluciones con programación dinámica para encontrar el espacio óptimo. Estos algoritmos consideran un número diferente de soluciones candidatas (que constituyen el espacio de búsqueda) al elegir el mejor plan. El resultado final es una serie de acciones, junto con su justificación y beneficio esperado para producir un plan significativamente mejor al inicial. Usualmente, se representa en forma de árbol.

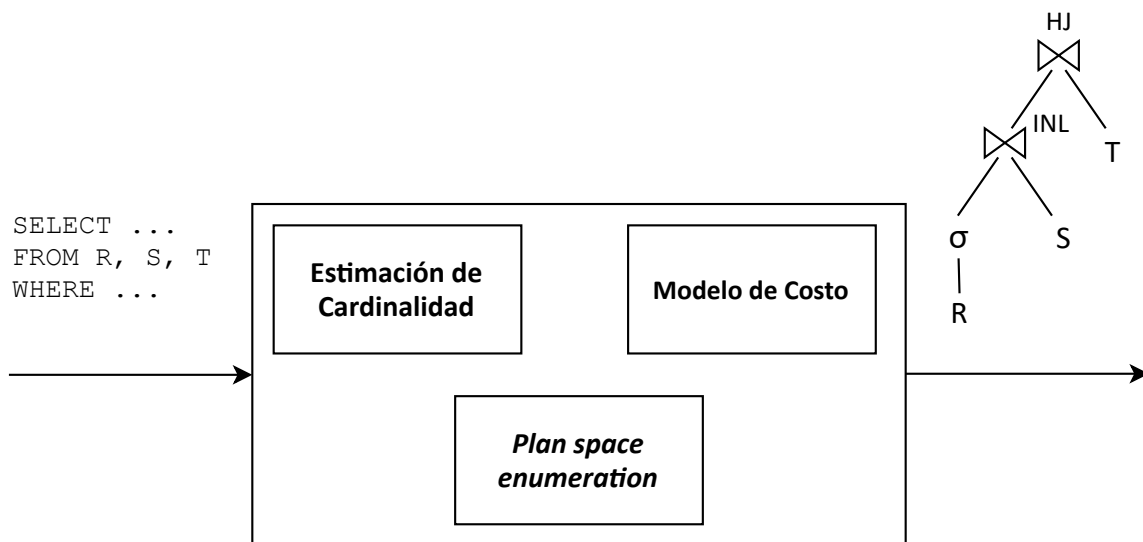


Figura 1: Arquitectura de un optimizador de consultas con enfoque basado en costo.

Fuente: [Leis et al., 2015].

2.1.2. Resource Description Framework - RDF

Una base de datos orientada a grafo (en inglés, *Graph Database*, o GDB) es un tipo de base de datos NoSQL. Como dice su nombre, utilizan una estructura de grafos para representar y modelar los datos, donde un conjunto de nodos o vértices representan entidades del dominio y los arcos o aristas representan las relaciones entre estas entidades. Un grafo *Resource Description Framework* (RDF), es un modelo de GDB, y es el estándar recomendado por el consorcio W3C para representar información en la *World Wide Web*, siendo clave en la Web Semántica. A continuación se detallarán algunos conceptos claves para comprender los grafos RDF [W3C et al., 2014a] [W3C et al., 2014b] [Hasan y Gandon, 2014].

Términos RDF

- Los **Internationalized Resource Identifier**, o simplemente **IRIs**, representan una cadena *Unicode* dentro de un grafo RDF. En otras palabras, representan un recurso. Los IRIs en esencia, son simplemente una generalización de las URIs (*Uniform Resource Identifier*).
- Los **Literales** representan valores como *strings*, números o fechas, lo que implica que están directamente asociados a un tipo de dato.
- Los **Nodos blancos** son nodos disjuntos (que no tiene ninguna relación) con los IRIs y los literales. El conjunto de posibles nodos blancos es arbitrario. RDF no hace referencia a ninguna estructura interna de nodos en blanco. No forman parte de la sintaxis abstracta de RDF, pero dependen completamente de la sintaxis.

RDF-triple

Sea I un conjunto de IRIs, L un conjunto de literales, y B un conjunto de nodos blancos. Un RDF-triple (s, p, o) es un miembro del conjunto $(I \cup B) \times I \times (I \cup L \cup B)$, donde s significa sujeto, p significa predicado, y o significa objeto, que son las componentes de un RDF-Triple (o simplemente triple). Se debe tener en cuenta que:

- el **sujeto** puede ser IRI o un nodo blanco.
- el **predicado** es un IRI.
- el **objeto** puede ser un IRI, un literal o un nodo en blanco.

Grafo RDF

Un grafo RDF es un conjunto de RDF-triples. Se puede visualizar como un nodo y un diagrama de arco dirigido, en el que cada triple se representa como un enlace nodo-arco-nodo. El sujeto y el objeto representan dos recursos relacionados, y el predicado representa la naturaleza de la relación, como se puede ver en la Figura 2.



Figura 2: Un RDF-triple con dos nodos (Bob y Alice) y un IRI que los conecta (*is a friend of*). En este caso, Bob es el sujeto, *is a friend of* es el predicado y Alice es el objeto.

Fuente: [W3C et al., 2014a].

Ejemplo sencillo de Grafo RDF

El siguiente ejemplo se puede visualizar muy bien cómo se construye un grafo RDF [W3C et al., 2014b]. Supongamos el siguiente conjunto de triples escritos en pseudocódigo:

```
<Bob><is a><person>.
<Bob><is a friend of><Alice>.
<Bob><is born on><the 4th of July 1990>.
<Bob><is interested in><the Mona Lisa>.
<the Mona Lisa><was created by><Leonardo da Vinci>.
<the video 'La Joconde à Washington'><is about><the Mona Lisa>
```

Es común hacer referencia al mismo recurso en varios triples. En el ejemplo, Bob es sujeto de cuatro triples, y la Mona Lisa es el sujeto de uno y el objeto de dos triples. Esta capacidad de tener el mismo recurso de estar en la posición del sujeto de un triple y la posición del objeto de otro permite encontrar conexiones entre triples, que es una parte importante de RDF. La Figura 3, muestra el grafo RDF resultante del pseudocódigo.

Como se mencionó anteriormente, las IRI se utilizan para identificar recursos como documentos, personas, objetos físicos y conceptos abstractos. Por ejemplo, el IRI para Leonardo da Vinci en DBpedia es: http://dbpedia.org/resource/Leonardo_da_Vinci. El IRI para un video del INA (un repositorio de videos y archivos audiovisuales franceses) sobre la Mona Lisa titulado 'La Joconde's Washington' en Europeana es: <http://data.europeana.eu/item/04802/243....> Los IRI son identificadores globales, por lo que otras personas pueden volver a usar este IRI para identificar lo mismo. Muchas personas utilizan el siguiente IRI como una propiedad RDF para indicar una relación de conocimiento entre personas: <http://xmlns.com/foaf/0.1/knows>. También se pueden ver ejemplos de literales, ya que hay cadenas como "La Joconde" y fechas como "14 July 1990". Un nodo en blanco, podría ser el fondo de la foto de la Mona Lisa, porque este representa un recurso sin nombre.

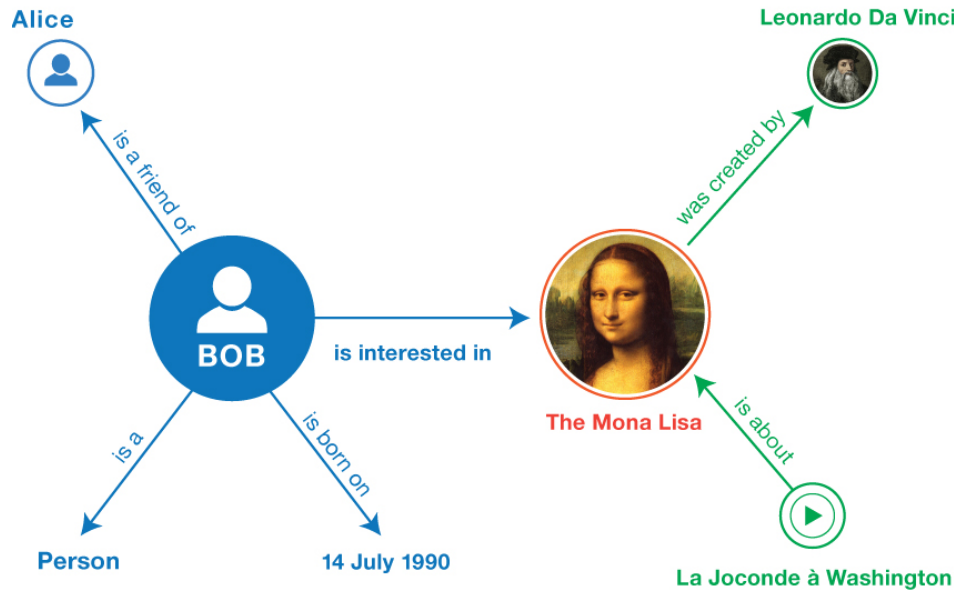


Figura 3: Grafo RDF generados por los triples del ejemplo.

Fuente: [W3C et al., 2014a].

2.1.3. SPARQL

SPARQL es el lenguaje de consultas estándar utilizado para RDF. Actualmente se utiliza su versión 1.1 [W3C et al., 2013]. Se define mediante una serie de pasos, comenzando desde la consulta SPARQL como un *string*, convirtiendo ese *string* en una forma de sintaxis abstracta, luego convirtiendo la sintaxis abstracta en una consulta abstracta de SPARQL que comprende operadores del álgebra de SPARQL. Esta consulta abstracta se evalúa en un conjunto de datos RDF. Como se describe en SPARQL 1.1 [W3C et al., 2013] [Hasan y Gandon, 2014], la resolución de consultas SPARQL se basa en la coincidencia o *mapeo* de patrones de grafo. Los motores de consulta SPARQL nativos realizan una serie de pasos para ejecutar una consulta, y que varían dependiendo el respectivo motor. No obstante todos tienen en común la realización de los siguientes pasos: Primero analizan el *string* consulta en un formulario de sintaxis abstracta. A continuación, transforman la sintaxis abstracta en consulta abstracta SPARQL. Finalmente, optimizan y evalúan la consulta abstracta SPARQL en un *dataset* RDF.

Un conjunto de datos RDF, o **RDF dataset**, se define como una colección de grafos RDF donde todos menos uno son grafos con nombre asociados con un IRI o un nodo blanco (el nombre del grafo) y el grafo predeterminado sin nombre [W3C y Zimmermann, 2014]. Mientras que una **consulta abstracta SPARQL** se define como una tupla (E, DS, QF) donde E es una expresión algebraica de SPARQL, DS es un *dataset* RDF y QF es la “forma de consulta” o *Query Form*.

La expresión algebraica E se evalúa contra grafos RDF en el *dataset* RDF DS . La forma de consulta QF utiliza las soluciones de las coincidencias de patrones para proporcionar

conjuntos de resultados o grafos RDF. Los QF disponibles:

- **SELECT**: Devuelve todas, o un subconjunto de, las variables enlazadas en una coincidencia de patrón de consulta.
- **CONSTRUCT**: Devuelve un grafo RDF construido mediante la sustitución de variables en un conjunto de plantillas triples.
- **ASK**: Devuelve un valor booleano que indica si un patrón de consulta coincide o no.
- **DESCRIBE**: Devuelve un grafo RDF que describe los recursos encontrados.

La expresión algebraica E incluye patrones de grafo y operaciones *statement* clásicas de los gestores de base de datos como **FILTER**, **JOIN**, **LEFT JOIN**, **UNION**, **ORDER BY** y **LIMIT** [W3C et al., 2013].

Un **patrón triple** (o simplemente triple cuando se está en un contexto de SPARQL) es un miembro del conjunto $(T \cup V) \times (I \cup V) \times (T \cup V)$, donde T es el conjunto de términos RDF (es decir, el conjunto $I \cup L \cup B$), y V es el conjunto de variables infinito y disjunto de T . En otras palabras, es un RDF-triple donde el sujeto, predicado y objeto pueden ser variables. **SPARQL permite formar patrones de grafos combinando patrones triples**, estos son los patrones de grafo básicos, en inglés, **Basic Graph Pattern** (a partir de ahora, simplemente **BGPs**), los cuales pueden ser patrones de grafos grupales (*Group Graph Pattern*), patrones de grafos opcionales (*Optional Graph Patterns*), entre muchos otros más. En la Figura 4 se puede ver un ejemplo de una consulta SPARQL en el *dataset* RDF de Wikidata, con cuatro patrones triples (dos de ellos con nodos en blanco) y dos BGPs, uno de ellos del tipo **OPTIONAL**.

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>

SELECT DISTINCT ?var1 ?var2 ?var3 ?var4
WHERE
{
  ?var1 wdt:P31 wd:Q11303 ;
        ?var3 wd:Q1297 ;
        wdt:P625 ?var2
  OPTIONAL
  {
    ?var1 wdt:P2046 ?var4
  }
}
```

Figura 4: Ejemplo de consulta SPARQL.
Fuente: Elaboración propia.

2.1.4. Machine Learning y Deep Learning

A continuación, se hará una breve introducción al aprendizaje automático o *Machine Learning* (ML), los paradigmas más importantes, y los pasos para hacerlo. También se explicará brevemente el concepto *Deep Learning* (DL).

Para comprender el funcionamiento y los objetivos que persigue el enfoque ML, es útil compararlo con el enfoque de la ingeniería de software tradicional [Ajanki, 2018]. Los ingenieros de software utilizan su ingenio humano para encontrar una solución y formularla como un programa preciso que una máquina puede ejecutar. Los científicos de datos, o simplemente personas que implementan sistemas de ML, no intentan escribir un programa por sí mismos. En cambio, recopilan datos de entrada y datos de salida (que son los valores objetivos deseados). Luego, le indican a una “máquina” que busque un algoritmo que calcule una salida para cada valor de entrada, para posteriormente, poder emplear ese algoritmo como herramienta de predicción o clasificación de datos. En otras palabras, el aprendizaje automático se trata de extraer patrones estadísticos a partir de datos. En la Figura 5 se clarifica lo recién explicado.

Para utilizar ML hay que seguir una serie de pasos [van Rijmenam, 2019] [Google, nd]:

1. *Obtener los datos*: El primer paso consta de obtener los datos en bruto. Estos deben ser la mayor cantidad de datos posible. Deben ser tanto datos de entrada como de salida (estos últimos, usualmente llamados *targets*).

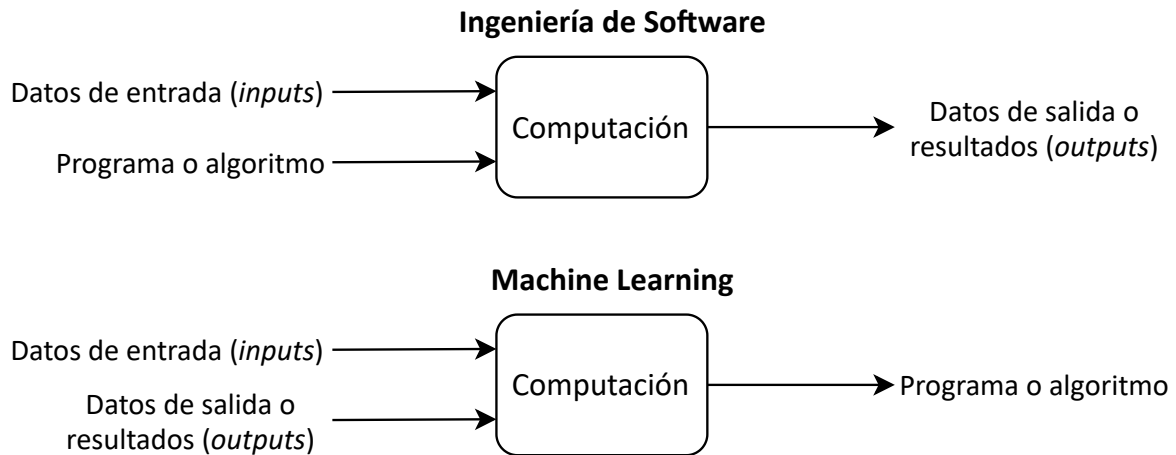


Figura 5: Comparación entre Ingeniería de Software y Machine Learning.

Fuente: [Ajanki, 2018].

2. *Procesamiento de los datos*: Los datos brutos o *raw data* no sirven de mucho por sí solo, por lo que se debe preparar y preprocesar los datos. Se debe normalizar, eliminar duplicados, eliminar datos extraños, etc.

En este paso, también se debe hacer la **caracterización o representación de los datos**, esto es, crear un vector de datos de características que representen los datos. **Es el primer gran desafío** a la hora de diseñar un modelo de ML. A este proceso también se lo conoce como ingeniería de características. Por ejemplo, supongamos una imagen de 128×128 píxeles. Como es sabido, esto se puede representar inicialmente como una matriz $(128, 128)$, en donde cada valor es una tupla con valores (R, D, B) , que representa la tonalidad del color de ese pixel. Entonces, esa matriz la puedo dividir por 255 para normalizarla, y posteriormente, hacerle un *reshape* (un remodelamiento), convirtiéndolo en una matriz de $1 \times (128 \cdot 128 \cdot 3)$, es decir, un gran vector que en este caso, sería el de características.

Por último, también se debe definir cuál será la cantidad de datos usados para el conjunto entrenamiento, cuántos serán usados para el conjunto de prueba y opcionalmente, se puede asignar una cantidad de datos para un conjunto de validación (usualmente se usa el conjunto de pruebas tanto para evaluar los resultados del conjunto de entrenamiento, como para probar el modelo, pero en muchas ocasiones es conveniente tener un conjunto de validación para evaluar los resultados del modelo y usar solo el conjunto de prueba para predicciones).

3. *Escoger el modelo*: El tercer paso consiste en seleccionar el modelo correcto, a su vez, se le considera el **segundo gran desafío** a la hora de diseñar un modelo de ML. Existen muchos modelos que se pueden usar para propósitos diferentes. Al seleccionar el modelo, se debe asegurar que este cumpla con el objetivo propuesto, es decir, que cumpla con el *inductive bias*². Además, se debe saber cuánta preparación requiere el modelo, cuán preciso es y cuán escalable es el modelo. Un modelo más complejo no siempre constituye un modelo mejor. Los algoritmos de aprendizaje automático comúnmente utilizados incluyen regresión lineal, regresión logística, *decision trees*, *K-means*, análisis de componentes principales (PCA), *Support Vector Machines* (SVM), Bayes, *Random Forests* y diversos tipos de redes neuronales.
4. *Entrenamiento, evaluación, ajuste de parámetros y predicción*: Son de los pasos más importantes del ML, a su vez, es el **tercer gran desafío** cuando se diseña un modelo de ML. Primero se comienza con el entrenamiento, donde el objetivo es utilizar los datos de entrenamiento y mejorar gradualmente las predicciones del modelo. Cada ciclo de actualización de los **parámetros del modelo** es un paso de entrenamiento. Por ejemplo, si quiero encontrar un algoritmo con la forma $f(x, y) = w_1x + w_2y + b$, entonces en cada paso se actualizarán los pesos w_1 , w_2 y el *bias* b . Posteriormente, se evalúa el modelo entrenado en un conjunto de datos (puede ser el de prueba o el de validación dependiendo del caso) no utilizados para ver cómo funciona. Esto podría ser representativo de cómo funciona el modelo en el mundo real, pero esto no tiene por qué ser así. En este paso actúa la función de optimización para comparar el resultado esperado, con el resultado obtenido. Una vez completado el entrenamiento y la evaluación, viene el momento de ajustar los hiperparámetros. El **tuning de hiperparámetros** (en inglés *hyperparameter tuning*), consiste en que, después de evaluar su modelo, se prueban y modifican los hiperparámetros establecidos originalmente para mejorar el modelo. Esto es por ejemplo, aumentar la cantidad de iteraciones, cambiar la tasa de aprendizaje, o aumentar las capas ocultas en caso de utilizar una red neuronal. Sin embargo, se debe definir cuándo un modelo es lo suficientemente bueno, de lo contrario, se podría llegar al sobreajuste, más conocido como *overfitting*, lo que provoca que el modelo solo pueda aprender de casos particulares y sea incapaz de reconocer nuevos datos, o por el contrario, llegar a un modelo *underfitting*, es decir un modelo que simplemente es poco preciso y poco o nada representativo y útil. El proceso de ajustar los hiperparámetros, es un proceso experimental y depende mucho de la habilidad de la persona con el manejo de datos. Finalmente, tras haber pasado por el proceso de recopilar y preparar los datos, seleccionar el modelo, entrenar y evaluar el modelo, y terminar de ajustar los hiperparámetros, es hora de responder preguntas y cumplir los objetivos que uno tenga, utilizando predicciones. Estos pueden ser todo tipo de predicciones, que van desde el reconocimiento de imágenes hasta la semántica y el análisis predictivo.

²En ML, el término *inductive bias* se refiere a un conjunto de suposiciones (explícitas o implícitas) hechas por un algoritmo de aprendizaje para realizar la inducción, es decir, generalizar un conjunto finito de observación (datos de entrenamiento) en un modelo general del dominio [Hüllermeier et al., 2013].

Los tipos o paradigmas de ML más utilizados son los siguientes [Simeone, 2018]:

- *Aprendizaje Supervisado*: Es el tipo de ML más básico y uno de los más utilizados. En el aprendizaje supervisado, el conjunto de entrenamiento consiste en pares de entrada y salidas esperadas, y el objetivo es aprender un mapeo entre los espacios de entrada y salida.
- *Aprendizaje No Supervisado*: En este tipo de ML, el conjunto de entrenamiento consiste en entradas no etiquetadas, es decir, entradas sin ninguna salida deseada asignada. El aprendizaje no supervisado generalmente apunta a descubrir las propiedades del mecanismo que genera los datos.
- *Reinforcement Learning*: El aprendizaje por refuerzo o *Reinforcement Learning* (RL) se encuentra, en cierto sentido, entre el aprendizaje supervisado y el no supervisado. A diferencia del aprendizaje no supervisado, existe alguna forma de supervisión, pero esto no viene en la forma de la especificación de una salida deseada para cada entrada en los datos. En cambio, un algoritmo RL recibe retroalimentación del entorno solo después de seleccionar una salida para una entrada u observación dada. La retroalimentación indica el grado en que la salida, conocido como acción en RL, cumple los objetivos del “alumno aprendiz” o “*learner*”, mas comúnmente llamado agente. RL se aplica a los problemas de toma de decisiones secuenciales en los que el agente interactúa con un entorno al tomar acciones secuenciales, las salidas, sobre la base de sus observaciones, sus entradas, mientras recibe comentarios sobre cada acción seleccionada. Más adelante se adentrará más en este concepto.

Deep Learning (DL) es un subcampo de ML relacionado con algoritmos inspirados en las redes neuronales artificiales. Las técnicas convencionales de ML generalmente, tienen una capacidad limitada para procesar datos naturales en su forma original. El aprendizaje de representación es un conjunto de métodos que permite que una máquina se alimente con datos en bruto y descubra automáticamente las representaciones necesarias para la detección o clasificación. Los métodos de DL son métodos de aprendizaje de representación con múltiples niveles de representación, obtenidos mediante la composición de módulos simples pero no lineales que transforman la representación que tiene cierto nivel (comenzando con la entrada sin formato) en una representación con nivel más alto, pero un poco más abstracto. Con la composición de suficientes transformaciones de este tipo, pueden aprender funciones muy complejas [LeCun et al., 2015].

Particularmente DL resulta útil para las tareas de clasificación, las capas superiores de representación amplifican aspectos de la entrada que son importantes para la discriminación y suprimen variaciones irrelevantes. No obstante, también se puede utilizar para regresión lineal. Al igual que ML, DL puede ser supervisado, no supervisado o de tipo *Reinforcement Learning*. En síntesis, la principal diferencia entre ML y DL es la forma de **presentar los datos**, ya que mientras en ML la representación de los datos requieren de una fuerte estructura, en DL generalmente dependerá de las capas de la/las red/es neurona/les.

2.1.5. Proceso de decisión de Markov

Un **proceso de decisión de Markov**, en inglés, **Markov Decision Process**, o simplemente MDP, en matemáticas, es un proceso de control estocástico en tiempo discreto. Proporciona un marco matemático para modelar la toma de decisiones en situaciones en las que los resultados son en parte aleatorios y en parte están bajo el control de quien toma las decisiones. Los MDP son útiles para estudiar problemas de optimización resueltos mediante programación dinámica[Bellman, 1957], siendo también muy útiles para diseñar modelos de RL precisamente por esto[Sutton y Barto, 2018]. A grandes rasgos, funcionan de la siguiente manera: en cada paso de tiempo, el proceso se encuentra en algún estado s , y el tomador de decisiones puede elegir cualquier acción a que esté disponible en el estado s . El proceso responde en el siguiente paso de tiempo moviéndose aleatoriamente a un nuevo estado s' , y dando al tomador de decisiones una recompensa correspondiente $R_a(s, s')$. La probabilidad de que el proceso pase a su nuevo estado s' está influenciada por la acción elegida. Específicamente, está dada por la función de transición de estado $P_a(s, s')$. Por lo tanto, el siguiente estado s' depende del estado actual s y de la acción del tomador de decisiones a . Pero dado s y a , es condicionalmente independiente de todos los estados y acciones anteriores; en otras palabras, las transiciones de estado de un MDP satisfacen la propiedad de Markov. En la Figura 6 se puede apreciar un ejemplo sencillo de un MDP. Este MDP representa un robot que puede recojer latas en una habitación. No obstante también debe recargar su pila cuando se le está acabando. Consta de dos estados $S = \{\text{Batería Alta}, \text{Batería Baja}\}$. Existen tres acciones posibles $A = \{\text{Esperar}, \text{Buscar Recargar}\}$, pero se observa que solo se puede acceder a la acción Recargar desde el estado Batería Baja, es decir $A(\text{Batería Alta}) = \{\text{Esperar}, \text{Buscar}\}$ y $A(\text{Batería Baja}) = \{\text{Esperar}, \text{Buscar}, \text{Recargar}\}$. Las recompensas son cero la mayor parte del tiempo, pero se vuelven positivas cuando el robot asegura una lata vacía, o grandes y negativas si la batería se agota por completo. Luego, en la misma figura se observa α y β que no son más que hiperparámetros que se ajustan manualmente para asignar la probabilidad de que el robot escoja alguna acción. Una explicación más detallada se puede hallar en [Sutton y Barto, 2018].

2.1.6. Reinforcement Learning

El *aprendizaje por refuerzo*, en inglés **Reinforcement Learning**, o simplemente RL, es un paradigma particular de ML, en donde se tiene un enfoque computacional para comprender y automatizar el aprendizaje dirigido a objetivos y la toma de decisiones, teniendo como base la programación dinámica. Se distingue de otros tipos de ML u otros aprendizajes computacionales, por su énfasis en el aprendizaje por parte de un agente a partir de la interacción directa con su entorno, sin requerir una supervisión ejemplar o modelos completos del entorno. El aprendizaje por refuerzo utiliza el marco formal de los MDP para definir la interacción entre un agente de aprendizaje y su entorno en términos de estados, acciones y recompensas. Este marco pretende ser una forma sencilla de representar las características esenciales del problema de la inteligencia artificial. Estas características incluyen un sentido

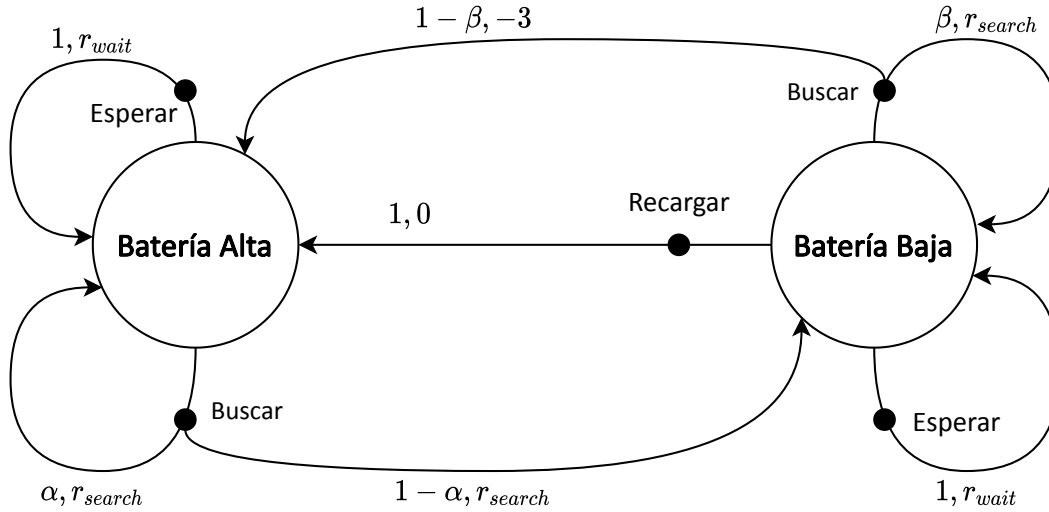


Figura 6: Ejemplo de MDP.
Fuente: [Sutton y Barto, 2018].

de causa y efecto, un sentido de incertidumbre y no determinismo, y la existencia de objetivos explícitos [Sutton y Barto, 2018]. Para diseñar un modelo de RL, es importante:

- Identificar y expresar el problema como un MDP, con todos los elementos principales de este: conjunto de acciones y estados, definición de recompensas y políticas.
- Escoger una buena **función de valor** o **value function** que permita que el agente aprenda de la mejor manera posible.

El **Deep Reinforcement Learning** o DRL, sería en resumidas cuentas, la mezcla entre RL y DL. Por ejemplo, un modelo DRL podría ser el uso de redes neuronales profundas para la elección de acciones, o para definir la recompensa.

2.1.7. Elementos de RL

- **Agente:** Es un elemento base del modelo, y es el que debe aprender. Por lo tanto, debe ser capaz de interactuar con su entorno hasta cierto punto, es decir, capaz de tomar acciones para pasar a distintos estados. El agente también debe tener un objetivo o meta relacionado a un estado del ambiente.
- **Entorno:** Es el mundo en donde el agente existe e interactúa.
- **Estado:** Es literalmente, algún estado presente en el entorno. Pueden ser terminales o no. Usualmente, se habla de un conjunto de estados $s \in S$.

- **Acciones:** Es literalmente una acción que el agente toma para pasar de un estado $s \in S$ a un estado $s' \in S$. En general, se habla de un conjunto de acciones $a \in A$. La acción puede ser escogida por el agente de distintas maneras. Por ejemplo, a través de un *greedy choice*, un ϵ -greedy o una red neuronal entrenada para elegir la acción.
- **Recompensa:** Es un valor numérico que se define en base al objetivo del problema RL. En cada paso, el entorno enviará una recompensa al agente. El objetivo del agente es maximizar la recompensa.
- **Política:** Una política define la forma de comportarse del agente en un momento dado. En términos generales, una política es un mapeo que va desde los estados percibidos del entorno hasta las acciones que se deben tomar cuando se está en esos estados. En algunos casos, la política puede ser una función simple, una matriz o una tabla de búsqueda, mientras que en otros puede implicar un cálculo extenso, como un proceso de búsqueda. La política es el núcleo de un agente de RL en el sentido de que por sí sola es suficiente para determinar el comportamiento.
- **Función de valor:** Mientras que la recompensa indica lo que es bueno en un sentido inmediato, una función de valor especifica lo que es bueno a largo plazo. En términos generales, el valor de un estado es la cantidad total de recompensa que un agente puede esperar acumular en el futuro, comenzando desde ese estado. Mientras que las recompensas determinan la deseabilidad intrínseca e inmediata de los estados del entorno, los valores de la función de valor indican la deseabilidad a largo plazo de los estados después de tener en cuenta los estados que probablemente seguirán y las recompensas disponibles en esos estados. Por ejemplo, un estado siempre puede generar una recompensa inmediata baja, pero aun así tener un valor alto porque es seguido regularmente por otros estados que brindan recompensas altas y viceversa.

2.1.8. Flujo general de un problema RL

En primer lugar, se definen los valores de los hiperparámetros iniciales, los cuales son propios de los algoritmos, como por ejemplo, un valor de tasa de aprendizaje α que poseen muchos algoritmos de RL, un factor de descuento γ que puede apoyar el aprendizaje del agente, el valor de ϵ en caso de que se tenga un ϵ -greedy para escoger las acciones del agente en cada paso, etc. Posteriormente, se plantea una política inicial $Q(s, a)$ (puede ser aleatoria o predefinida con anterioridad) del total de políticas Q generados por el mapeo de estados y acciones posibles. Luego se coloca el agente en un estado inicial $s_{initial} \in S$. A partir de ahí el agente escoge una acción que puede o no estar basada en la política inicial. El agente pasa a estar en un nuevo estado $s \in S$ que es distinto al inicial, y recibirá su recompensa R de acuerdo al nuevo estado en que se encuentre. Una vez escogida esta acción, se evalúa el valor de la política en ese estado y esta acción, es decir se evalúa $Q(s, a)$, utilizando alguna función de valor. Luego, se escoge una acción y se repite el proceso hasta que el estado $s \in S$ sea un estado terminal. El resultado será un nuevo conjunto de políticas Q que contendrá

un mejor resultado a seguir para el problema MDP planteado. Para aprender y mejorar el resultado final, se itera varias veces. En la Figura 7 se aprecia un diagrama simple de como el agente y el entorno interactúan en una iteración, y en la Figura 8 se puede observar un pseudocódigo de *Q-Learning*, una de las técnicas de RL más populares.

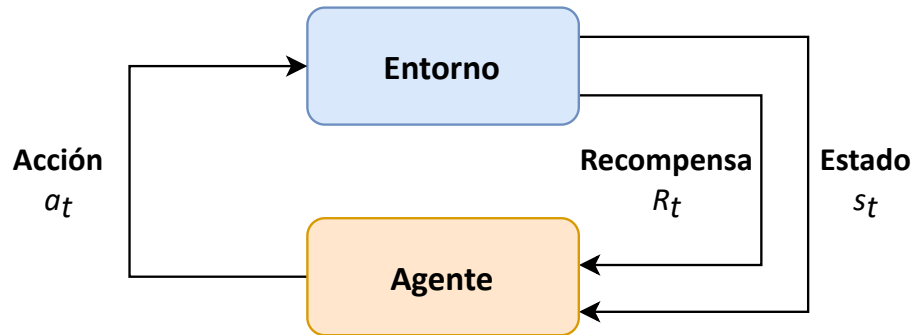


Figura 7: Diagrama de RL donde se refleja la interacción entre Agente y Entorno.

Fuente: [Amiri et al., 2018].

Hiperparámetros de entrada: $\alpha \in (0, 1), \varepsilon \in (0, 1), \gamma \in (0, 1)$
 Se inicia $Q(a, s)$, para todo $s \in S, a \in A$
Loop para cada iteración:
 Se inicializa con algún $s \in S$ inicial
 Loop para cada paso de una determinada iteración, mientras $s \in S$ no es **terminal**
 Se escoge un $a \in A$ de S usando alguna política derivada de Q (por ejemplo, ε -greedy);
 Una vez tomada la acción $a \in A$, se obtiene la recompensa R y se pasa al nuevo estado $s' \in S$
 Se calcula la función de valor a través del algoritmo de Q-Learning, el cual se calcula como:
 $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma \max_a Q(s', a) - Q(s, a)]$
 $s \leftarrow s'$, el estado actual pasa a ser el nuevo estado recién obtenido

Figura 8: Pseudocódigo de algoritmo *Q-Learning*, con algunos comentarios que indican que se está haciendo.

Fuente: [Sutton y Barto, 2018].

2.2. Herramientas

2.2.1. Virtuoso Open Source

OpenLink Virtuoso Universal Server es una base de datos SQL relacional de objetos de alto rendimiento. Existe una versión pagada, así como la versión de código abierto **Virtuoso Open Source** (a partir de ahora, solo VOS), donde esté último es el que se utilizará como gestor de base de datos para la memoria. Como base de datos, proporciona transacciones, un compilador inteligente de SQL, un potente lenguaje de procedimiento almacenado con alojamiento opcional en el lado del servidor Java y .Net, copia de seguridad en caliente, compatibilidad con SQL-99, servidor web integrado, repositorio WebDAV integrado, y más. Tiene las principales interfaces de acceso a datos, como ODBC, JDBC, ADO .Net y OLE / DB. En cuestión, tiene muchas características, propiedades y funcionalidades interesantes, pero la funcionalidad más interesante que compete a esta memoria, es que permite almacenar datos en formato RDF, así como también utilizar consultas SPARQL en estos. Es decir, permite en un ambiente de base de datos SQL utilizar consultas SPARQL. Para lograr esto realiza varias tablas índices y funciones internas, que en resumidas cuentas "traduce" una consulta SPARQL a una SQL, como también "traduce" el almacenamiento de datos RDF no relacional. Para más información de esto ver sección 16.5 (particularmente las subsecciones 16.5.3, 16.5.4 y 16.5.5) de la documentación [OpenLinkSoftware, 2021]. En general en el momento de la traducción los triples se suelen unir con sentencias SQL del tipo `INNER JOIN`, a excepción de que estos triples se encuentren dentro de una sentencia `OPTIONAL`, en donde se unen a través de un `LEFT OUTER JOIN`. En la Figura 9 se puede ver la traducción a SQL que hace VOS de la consulta mostrada en la Figura 4. En esta figura, se observa el uso de varias tablas índices auxiliares como `DB.DBA.RDF_QUAD`, así como también de funciones internas como `__id2in` o `__ro2sq`. También se observa como al momento de traducir la sentencia `OPTIONAL` lo hace a través de un `LEFT OUTER JOIN`.

La ventaja que utilizará la memoria de la traducción, es que VOS permite utilizar sentencias típicas de gestores de bases de datos SQL en consultas SPARQL, como por ejemplo `EXPLAIN` o `PROFILE`³). Estos permiten obtener el plan de ejecución completo de una consulta SPARQL, lo que significa muchos datos interesantes como el tiempo de ejecución total, el tiempo de ejecución por subconsulta, el orden en que se ejecutan estas subconsultas, si estas pertenecen a una declaración `OPTIONAL`, y más ⁴. Resumidamente, `PROFILE` se diferencia del `EXPLAIN` porque el primero entrega mucha más información que el segundo. Eso sí, lamentablemente no existe una herramienta de VOS que permita extraer características con facilidad, pues la salida solo lo envía a la terminal, por lo cual será necesario desarrollar un extractor desde cero. En el Capítulo 3 se explicará en detalle como leer la salida de una sentencia `PROFILE` y como se extraen características de este.

³How can I get full explain and profile plans for a simple SPARQL query? - VOS Documentation.

⁴6.1.7. Performance Tuning, Reading a Query profile - VOS Documentation.

```
SELECT __ro2sq ("s_2_9_rbc"."var1") AS "var1", __ro2sq ("s_2_9_rbc"."var2")
AS "var2", __ro2sq ("s_2_9_rbc"."var3") AS "var3", __ro2sq ("s_2_9_rbc"."var4")
AS "var4" FROM (SELECT DISTINCT "s_2_10_t6"."S" AS "var1",
    "s_2_10_t6"."O" AS "var2",
    "s_2_10_t5"."P" AS "var3",
    "s_1_8"."var4" AS "var4"
FROM DB.DBA.RDF_QUAD AS "s_2_10_t4"
    INNER JOIN DB.DBA.RDF_QUAD AS "s_2_10_t5"
    ON (
        "s_2_10_t5"."S" = "s_2_10_t4"."S")
    INNER JOIN DB.DBA.RDF_QUAD AS "s_2_10_t6"
    ON (
        "s_2_10_t6"."S" = "s_2_10_t5"."S"
        AND
        "s_2_10_t6"."S" = "s_2_10_t4"."S")
LEFT OUTER JOIN (
    SELECT "s_1_8_t0"."S" AS "var1",
        "s_1_8_t0"."O" AS "var4"
    FROM DB.DBA.RDF_QUAD AS "s_1_8_t0"
    WHERE
        "s_1_8_t0"."P" = __i2idn ( __bft(
            'http://www.wikidata.org/prop/direct/P2046',1))) AS "s_1_8"
ON (
    "s_2_10_t6"."S" = "s_1_8"."var1"
    AND
    "s_2_10_t5"."S" = "s_1_8"."var1"
    AND
    "s_2_10_t4"."S" = "s_1_8"."var1")
WHERE
    "s_2_10_t4"."P" = __i2idn ( __bft('http://www.wikidata.org/prop/direct/P31',1))
    AND
    "s_2_10_t4"."O" = __i2idn ( __bft('http://www.wikidata.org/entity/Q11303',1))
    AND
    "s_2_10_t5"."O" = __i2idn ( __bft('http://www.wikidata.org/entity/Q1297',1))
    AND
    "s_2_10_t6"."P" = __i2idn ( __bft('http://www.wikidata.org/prop/direct/P625',1))
OPTION (QUIETCAST)) AS "s_2_9_rbc"
```

Figura 9: Ejemplo de traducción de una consulta SPARQL a una consulta SQL en VOS. La consulta traducida es la presentada en la Figura 4.

Fuente: Elaboración propia.

2.2.2. Otras herramientas computacionales

Para la extracción y caracterización de consultas, se utilizará como lenguaje de programación **Python**. Se tomó esta decisión porque es un lenguaje útil para realizar scripts de "*data scrapping*", permitiendo extraer y manejar datos que se puedan obtener de VOS con relativa facilidad. Para apoyar la extracción, también se usaran algunos *scripts* en *bash*.

Además, para los modelos de ML y DL también se utilizará *Python*, ya que dispone de un gran soporte, una gran comunidad y de muchas librerías que sirven para utilizar diversas técnicas relacionadas. Para los modelos de la memoria, se usarán las librerías **Pandas**, **Numpy**, **Scipy**, **Pytorch** y *notebooks* de **Jupyter**.

2.2.3. Hardware

Para hacer las consultas y entrenar el modelo se utilizó un servidor externo con las siguientes características: AMD, Opteron Server con 24 cores, 2.8 GHz, 64 GB memoria RAM, tarjeta gráfica GeForce RTX 2080 con 2GB, CUDA Version:10.1. Linux Operating System version 5.0.0-37.

2.3. Estado del arte

En la literatura se pueden encontrar diversos trabajos relacionados a la optimización de *performances* para consultas en distintos gestores de BD. Estos trabajos pueden ser modelos que predicen el tiempo de ejecución de una consulta, o modelos aún más complejos que utilizan los modelos mencionados anteriormente para predecir un plan de consulta más apropiado y optimizado que el que podría dar un gestor de base de datos tradicional. La gran mayoría de estas técnicas propuestas utilizan SQL como lenguaje de consulta, pero existen algunos trabajos que usan SPARQL. En particular, hay tres trabajos que son influencia directa en la realización de esta memoria:

- **A Machine Learning Approach to SPARQL Query Performance Prediction [Hasan y Gandon, 2014]**

En este trabajo se realizó un modelo ML con el objetivo de predecir tiempos de ejecución de consultas SPARQL. El *dataset* utilizado fue uno basado en DBpedia 3.5.1. Para la caracterización de los datos se utilizaron las llamadas "características algebraicas de SPARQL"[W3C et al., 2007], las cuales son en palabras simples, características propias de una consulta (por ejemplo, si tiene o no una declaración JOIN, si tiene o no una declaración OPTIONAL, el número de BGPs y triples que posee, etc.), pero también se utilizaron características de "patrones de grafo" los cuales representan las similitudes

de los grafos (en este caso usaron la distancia de edición⁵) de cada consulta con respecto a K consultas más representativas del *dataset* de entrenamiento. Como modelos de regresión, se usaron los clásicos *Support Vector Machine* para (Nu-SVR) y los *K-Nearest Neighbors* para predecir el tiempo de ejecución de las consultas. Para la evaluación se utilizaron como métricas *R2 Score* y el la raíz cuadrada del error cuadrático medio. Los resultados no fueron los mejores debido, entre otras cosas, a que el *dataset* era demasiado pequeño. No obstante los mejores resultados se obtuvieron con Nu-SVR y usando tanto las características algebraicas como los patrones de grafo.

■ **Neo: A Learned Query Optimize [Marcus et al., 2019]**

Este trabajo, realizado para el lenguaje de consultas SQL, es uno de los más interesantes pues se utilizó un modelo de DRL con el objetivo de encontrar un plan de consultas mejor a varios optimizadores de consultas comerciales en el mercado, como PostgreSQL, SQLite, Oracle o MS SQL Server. Para lograr esto, propone un optimizador de consultas con soluciones *end-to-end*. El aprendizaje se hace a través de RL. La caracterización de datos de las consultas se clasifican en dos categorías: información a nivel de consulta (la consulta completa, donde sus características se codifican en estructuras de datos como vectores *one-hot* que representan características generales, histogramas y *R-Vectors*) e información a nivel de planes de ejecución (incluye características tanto de la consulta completa como de las subconsultas de estas, codificadas en arboles binarios de ejecución, donde se preserva la estructura del plan de consultas. Por ejemplo, contiene información sobre el orden de los *JOINS*). Con esta información, en primer lugar se entrena un modelo clásico de DL para la predicción del tiempo de ejecución de la consulta, utilizando una arquitectura de redes neuronales denominada *Neo's value network*, donde se tiene la particularidad de que se utilizaron redes neuronales del tipo **Tree Convolution**, una red neuronal inspirada en las *Convolutional Neural Network* o CNN, las cuales son capaces de procesar información en formato de árbol. También posee capas ocultas de redes neuronales comunes. La arquitectura propuesta es capaz de procesar tanto la información a nivel de consulta como la información a nivel de planes de ejecución. En la Figura 10 se aprecia un diagrama de la *Neo's value network*. Una vez entrenado el modelo DL para la predicción de tiempos de ejecución, cada consulta es modelada como un MDP, diseñando así un modelo RL, en donde un plan de consultas parcial es un estado, y cada subconsulta es una acción a tomar. La recompensa por su parte es el negativo de la predicción dada por la red neuronal entrenada. De esta forma, se propone primero una política inicial, que va mejorándose con varias iteraciones. Como se mencionó anteriormente, se va construyendo el árbol de ejecución (agregando alguna subconsultas que sería la acción, pasando a un nuevo plan de consultas parcial, es decir, un subárbol, el cual sería un nuevo estado) hasta lograr un árbol donde se representa el plan de ejecución de la consulta (estado terminal), y se evalúa en la *Neo's value network* para obtener la recompensa. Cada estado tendrá un valor en la política que no será en sí el mejor

⁵La distancia de edición es una forma de cuantificar la diferencia entre dos *strings* contando el número mínimo de operaciones necesarias para transformar un *string* en otro.

valor posible para ese estado (es decir, no se evalúan planes de ejecución parciales), más bien, es el mejor valor posible si se sigue ese estado. Una vez completada la política se evalúa nuevamente en varias iteraciones. En la Figura 11 se aprecia una política completa, donde se ve cuál sería el valor de evaluación en cada estado (osea, en cada subárbol), junto al mejor plan de ejecución posible.

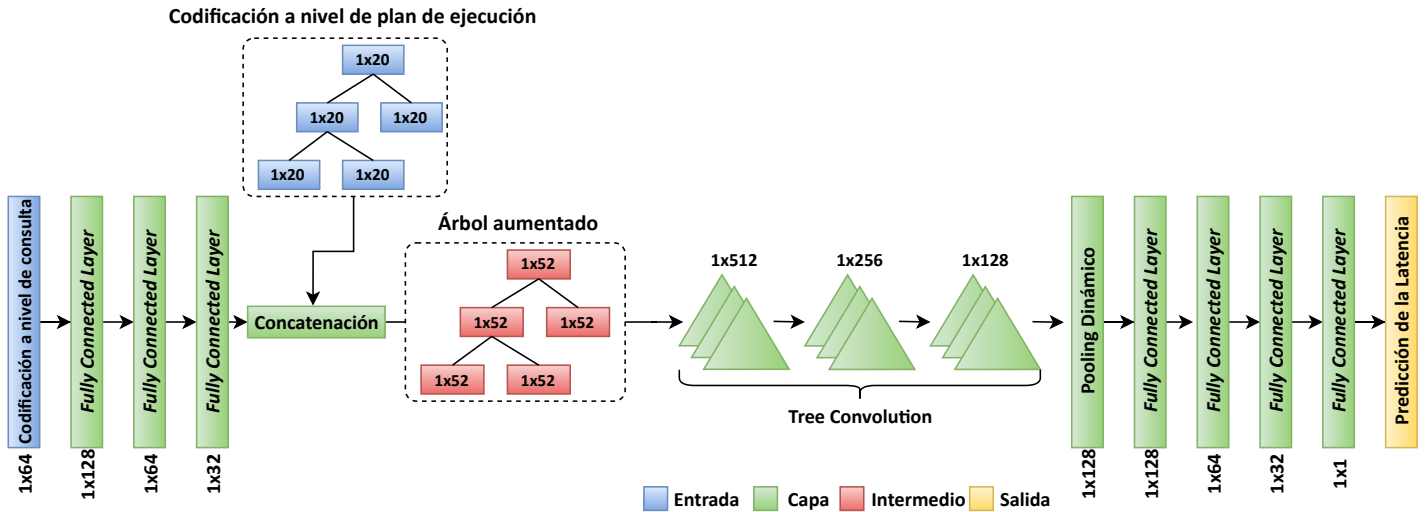


Figura 10: Diagrama de la Neo's value network.

Fuente: [Marcus et al., 2019].

■ Predicción de rendimiento en consultas SPARQL con Deep Neural Network [Casals et al., 2020]

En este trabajo se tuvo como objetivo predecir los tiempos de ejecución de una consulta SPARQL, y es interesante pues se unen conceptos de los otros dos trabajos mencionados anteriormente. Por un lado, de [Hasan y Gandon, 2014] se utilizaron algunas codificaciones para la caracterización de consultas. De parte de Neo, se hizo una versión diseñada para SPARQL del modelo con redes neuronales *Tree Convolution*. De esta forma, las características se representaron teniendo datos a nivel de consulta (similares a los de [Hasan y Gandon, 2014] como podrían ser la representación algebraica y los patrones de grafo, pero también se agregaron otras representaciones como los histogramas). También se usaron datos a nivel de planes de ejecución, en donde propuso una estructura de árbol, los cuales contenían la información de los tipos de JOINS similar a lo propuesto en [Marcus et al., 2019], y además se incluyeron características propias de SPARQL como el tipo de triple, los predicados participantes, entre otros. En la Figura 12 se puede ver la representación en formato árbol. En la figura, se observa como se va codificando información como el predicato que actúa en un determinado subárbol o consulta individual, así como también los tipos de JOINS y tipos de triples. Para la evaluación del modelo se usó el Error Cuadrático Medio, la Raíz del Error Cuadrático Medio y el Error Absoluto Medio. También se hizo una prueba de hipótesis *t-student* para asegurarse que los resultados no fueron aleatorios.

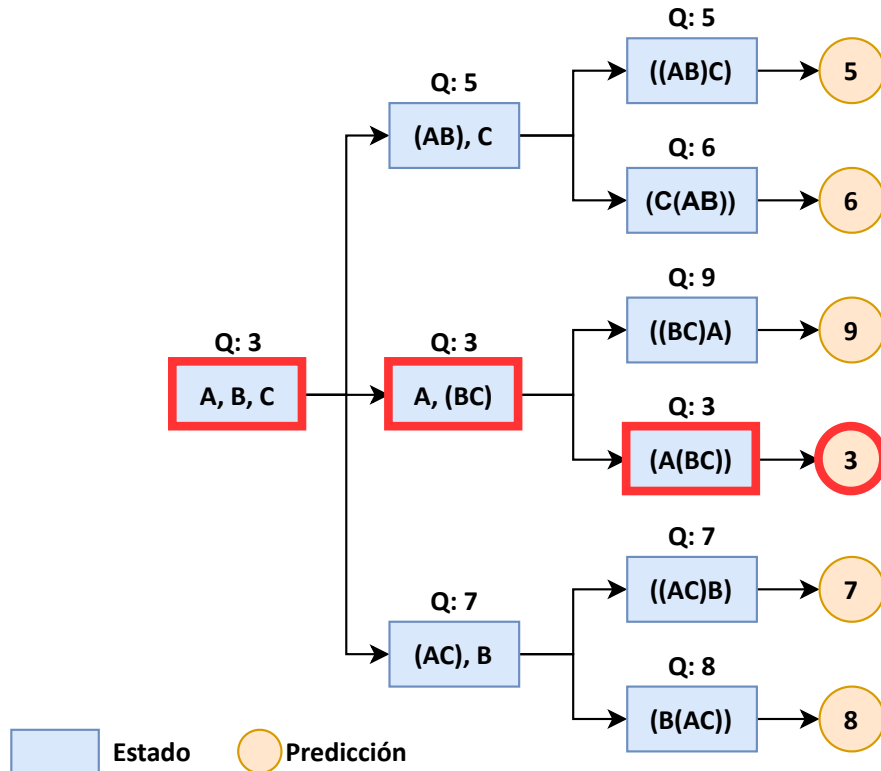


Figura 11: Diagrama de una política en Neo, donde se puede apreciar el valor en la política Q de cada estado. En rojo se remarca el mejor árbol, y por ende, plan de ejecución, posible.

Fuente: [Marcus et al., 2019]. En particular, fue presentado en *Boston University MiDAS Seminar* donde se expuso el proyecto Neo.

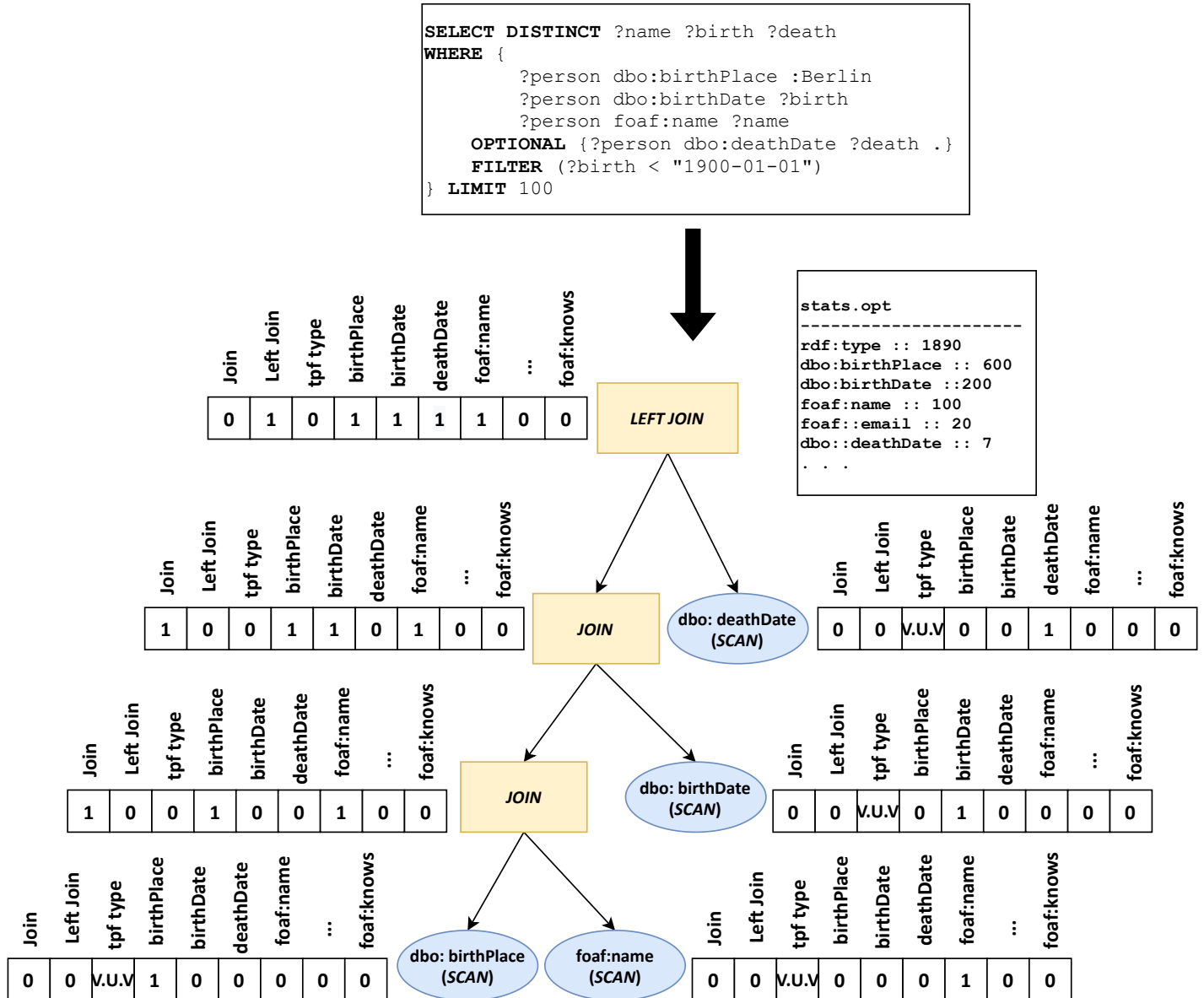


Figura 12: Estructura en formato árbol de una consulta SPARQL.

Fuente: [Casals et al., 2020].

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

La solución propuesta consta en modelos de **Deep Reinforcement Learning** que permiten encontrar un mejor plan de consultas, para un conjunto de consultas SPARQL. Se desarrollaron cuatro modelos de DRL que poseen distintos algoritmos de función de valor. Para explicar como se hicieron estos modelos, es necesario detallar como se resolvieron los desafíos de **extracción y caracterización de los datos** y el del **diseño del modelo**. Se utilizó como base principalmente los proyectos de [Hasan y Gandon, 2014], [Marcus *et al.*, 2019] y [Casals *et al.*, 2020]. Los modelos propuestos están diseñados para que construyan árboles binarios de consultas, los cuales representan planes de consultas optimizados.

3.1. Extracción y caracterización de los datos

Con respecto a la extracción, se propone una estrategia basada en “data scrapping”, en donde se utilizan varios *scripts* que ejecutan consultas, y leen distintos archivos y “outputs” dados por VOS, el gestor de base de datos escogido, y que posteriormente codifican la información dada en alguna estructura de datos. La caracterización de los datos se hizo **a nivel de consulta** y **a nivel de ejecución**. Ambas caracterizaciones tienen una representación distinta.

3.1.1. Extracción de los datos

La extracción de datos se hace, como se mencionó previamente, a través de varios *scripts* programados en *Python* y *bash*. En primer lugar, se ejecutan distintas consultas, pero no de forma normal, sino que se utiliza la sentencia `PROFILE`⁶, la cual permite obtener mucha más información tanto a nivel de consulta, subconsulta, planes de ejecución, etc. Esta sentencia genera una salida que luego se guarda en un archivo de texto. Al finalizar este primer paso, se tendrán varios archivos de texto `PROFILE` guardados. Posteriormente, se leen estos archivos y se detectan distintos patrones de texto en cada archivo `PROFILE`, los cuales en síntesis, son las características que se deseaban identificar. Una vez procesadas, se guardan en un archivo `.csv`. También se guardan todas los planes parciales en ese archivo. En la Figura 13 se puede ver un fragmento de una sentencia `PROFILE` ejecutado en la consulta de la Figura 4, junto con algunas anotaciones respecto a como detectar ciertos patrones. Un archivo `PROFILE` contiene varias “operaciones” internas, las cuales pueden identificarse porque cada una de estas comienza detallando datos de ejecución *time-fanout-row_input*. Estos operadores pueden significar muchas cosas dependiendo de la consulta, como por ejemplo cuando se ejecuta un `GROUP BY`, cuando se ejecuta un `LIMIT`, etc. Obviamente, también muestra cuando se

⁶How can I get full explain and profile plans for a simple SPARQL query? - VOS Documentation.

ejecuta una subconsulta en particular, las cuales pueden identificarse pues ocupan tablas índices destinadas a aquello como por ejemplo `RDF_QUAD_POGS`, y contienen un caracter `P` el cual representan el predicado. Opcionalmente, pueden contener los caracteres `S` y/o `O` los representan el sujeto y el objeto de esa subconsultas respectivamente. En caso de que no esté el sujeto y/o el objeto representado en el `PROFILE`, quiere decir que es una variable. Además, si estás consultas se encuentran bajo un operador `outer`, entonces estamos hablando de una consulta que está siendo ejecutada con un `LEFT OUTER JOIN`, es decir, de una sentencia `OPTIONAL`. Finalmente, se puede calcular el tiempo de ejecución de cada subconsulta sumando el tiempo de ejecución de todo los operadores anteriores de la subconsulta más el tiempo ejecución que tarde la subconsulta como tal, datos que VOS entrega. En la Figura 13 se remarcan en un cuadro celeste las subconsultas que se están ejecutando, además de remarcar en verde si están o no en una sentencia `OPTIONAL`. También se remarca en rojo el tipo de triple que representa cada subconsulta.

```

{
  time      0.0079% fanout          1 input          1 rows
    clear: (<V set_ctr$89 in>)
    ...

  time      0.64% fanout          54 input          1 rows
  RDF_QUAD_OP      20 rows(t3.P$34)
    inlined O = IRI_ID"...Q1297"

  time      11% fanout      185.185 input          54 rows
  Cardinality estimate:      1.2 Fanout:      1.9e+02      Triple del tipo VAR_VAR_URI
  RDF_QUAD_POGS      1.2 rows(t3.S$38)
    P = t3.P$34 , O = IRI_ID"...Q1297"

  time      41% fanout      0.0084 input      10000 rows
  Cardinality estimate:      6.2e-05 Fanout:      0.0084      Triple del tipo VAR_URI_URI
  RDF_QUAD_POGS      6.2e-05 rows(t2.S$42)
    inlined P = IRI_ID"...direct/P31", O = IRI_ID"...Q11303", S = t3.S$38

  time      28% fanout      0.988095 input          84 rows
  RDF_QUAD      1 rows(t4.S$46, t4.O$45)      Triple del tipo VAR_URI_VAR
    inlined P = IRI_ID"...direct/P625", S = k_t3.S$144

  time      0.035% fanout          1 input          83 rows
  outer {
    time      8% fanout      0.0240964 input          83 rows
    Cardinality estimate:      1.1 Fanout:      0.024      Triple del tipo VAR_URI_VAR
    RDF_QUAD      1.1 rows(t6.O$49)
      inlined P = IRI_ID"...direct/P2046" , S = k_t4.S$168
    time      0.02% fanout          41.5 input          2 rows
  } /* end of outer */
  ...

  time      0.0054% fanout          0 input          83 rows
  Select (var1$75, var2$77, var3$79, var4$81)
}

8 msec 71% cpu,      10269 rnd      10032 seq  97.955% same seg  0.877% same pg
Compilation: 4 msec 0 reads      0% read 0 messages      0% clw

```

Figura 13: Ejemplo de sentencia PROFILE ejecutado por VOS en la consulta de la Figura 4.

Fuente: Elaboración propia.

3.1.2. Caracterización a nivel de consulta

La caracterización a nivel de consulta registra en primer lugar, propiedades algebraicas de las consultas, es decir, la frecuencia con la que ciertos operadores actúan o no. En particular, se decidió registrar la cantidad de JOINS y LEFT JOINS presentes en las consultas. Además, también registra datos relacionados con los árboles de ejecución que representan el plan de ejecución de la consulta (más adelante se detallarán estos) y con el plan de ejecución propiamente tal, como lo son el tamaño del árbol, la cantidad de BGPs involucrados, etc. La estructura de datos para representar esta caracterización es simplemente un vector. Es importante tener en cuenta que las características a nivel de consulta deben ser representativas tanto para subconsultas como para consultas completas. La razón de esto es porque el modelo de predicción de latencia explicado más adelante, recibe tanto consultas como subconsultas para el entrenamiento, validación y test. Es por eso que se descartaron propiedades que pudieron ser interesantes, como los operadores GROUP BY u ORDER BY, ya que estos solo se ejecutaban al finalizar la consulta (salvo que existiera una sentencia UNION u otro caso similar), y que a la larga provocaban dificultades para entrenar el modelo. Aclarado este punto, las características a nivel de consulta son:

- **time** : Tiempo de ejecución de la consulta. En caso de que sea una subconsulta, se hace una pequeña transformación a unidad de tiempo ya que el dato viene inicialmente como porcentaje. También muy importante a tener en cuenta que es el **target** que se desea predecir y optimizar.
- **treesize** : Tamaño del árbol binario que representa el plan ejecución de la consulta.
- **triples** : Cantidad de triples involucrados en la consulta.
- **total_bgp** : Cantidad de BGPs involucrados en la consulta.
- **JOIN** : Cantidad de JOINS involucrados en la consulta.
- **LEFT JOIN** : Cantidad de LEFT JOINS involucrados en la consulta.

3.1.3. Caracterización a nivel de plan de ejecución

Las características a nivel de plan de ejecución registran los pasos que da una consulta para que esta se ejecute en el gestor de la base de datos, es decir, registran el **plan de consultas**. El plan de consultas en este caso se obtiene del archivo PROFILE. En otras palabras, la caracterización a nivel de consultas va mostrando como una consulta se va construyendo a través de varias otras subconsultas, o bien, como una subconsulta parcial se construye en base a otras subconsultas parciales. La información más importante de las consultas que se utilizarán en el modelo vienen de esta caracterización. La estructura de datos para representar estas características escogida es similar a la de [Casals *et al.*, 2020], es decir, se representarán en un

formato de árbol binario. Los nodos del árbol pueden representar triples, o sentencias JOINS o LEFT JOINS, en donde los primeros siempre serán nodos hojas, y los segundos siempre serán nodos intermedios, incluyendo la raíz. El proceso de construcción del árbol binario es el siguiente:

1. Se identifican y se extraen los triples de la consulta, y el orden en que estos se ejecutan, procesando el archivo PROFILE de la consulta. También se registra en cual BGP está el triple.
2. Una vez extraídos los triples, se va construyendo el árbol, primeramente **intra-BGP**. Se toman los primeros dos triples de un BGP y quedan como nodos hojas, los cuales guardarán además distintos datos que se detallarán más adelante. La raíz de estos dos nodos hojas será una sentencia JOIN.
3. Se repite el paso 2, pero ahora se elige el triple siguiente del BGP que se está procesando, y el subárbol construido previamente.
4. Se repiten los pasos 2 y 3 hasta que no queden BGPs que procesar.
5. Posteriormente, se comienzan a unir subárboles, lo que sería un procesamiento **entre-BGP**. Se unen los BGPs (en el orden dado por el plan de consultas) con una raíz que representa un JOIN, o un LEFT JOIN en caso de que algunos de los BGPs estén dentro de una sentencia OPTIONAL.
6. Se repite el paso 5 hasta que no queden BGPs por procesar.

Cada nodo a su vez, tendrá información representante de la subconsulta, los cuales son las características a nivel de plan de ejecución. Estas características son:

■ Nodos Hojas:

- **triple_type**: Es un *string* que representa el tipo de triple. Puede ser VAR_VAR_VAR, VAR_VAR_URI, VAR_URI_VAR, VAR_URI_URI, VAR_URI_LITERAL, VAR_VAR_LITERAL, URI_URI_LITERAL, URI_URI_VAR, URI_URI_URI, URI_VAR_VAR, URI_VAR_URI, URI_VAR_LITERAL, LITERAL_URI_VAR, LITERAL_URI_URI, LITERAL_URI_LITERAL. Posteriormente se codifica como un vector *one-hot*.
- **predicate**: Representa el predicado del triple. Primero se mapean todos los predicados que se van a utilizar de todo el *dataset*, y se les asigna un número. Luego, el predicado del nodo hoja se codifica con este número.

■ Nodos intermedios:

- **join_type**: Representa el tipo de JOIN entre dos nodos hijos del árbol. Es un valor categórico que tiene los valores JOIN y LEFT_JOIN, y que se codifica como un vector *one-hot* de largo dos.

- **child_predicates:** Representa los predicados que participan en la subconsulta, es decir, los predicados de los dos nodos hijos. Va aumentando a medida que se sube en el árbol, representándose en una suma. Por lo tanto, la raíz tendrá la suma de los predicados de todo los nodos hojas del árbol.

Los árboles del plan de consulta pueden ser de profundidad variable dependiendo de la cantidad de subconsultas que se tengan. Pero los vectores de los nodos son de tamaño fijo. La Figura 12 presentada en el Capítulo 1, es un diagrama que refleja muy bien como una consulta es representada a nivel de árbol.

3.1.4. Dataset

Una vez construido el extractor y definido las características a extraer, es importante definir el conjunto de datos, más conocido como **dataset**, que se utilizará, ya que es importante tener una cantidad de datos representativa para poder crear un modelo satisfactorio. Por ejemplo, en [Hasan y Gandon, 2014] tuvieron un resultado no muy concluyente debido a que precisamente el *dataset* no era lo suficientemente bueno. Pero no solamente la cantidad de datos importa, también es importante que los valores de los datos no sean extremadamente dispersos entre sí. De nada sirve tener un *dataset* que tenga 1.000 datos con valor 1, y 1.000 datos con valores cercanos 1.000.000. Es muy importante eliminar *outliners*, pero también es muy importante que el *dataset* sea lo más balanceado posible.

Respecto a las consultas SPARQL, existen pocos *dataset* disponibles, ya que como se ha mencionado anteriormente, la utilización de técnicas de ML en SPARQL no es un tema tan abordado aún. Por eso se propone la elaboración de un *dataset* con consultas de Wikidata, provenientes de:

- Wikidata Benchmark [Buil-Aranda y Rojas, 2020]
- Consultas utilizadas por [Casals et al., 2020]
- Las subconsultas totales generadas por las consultas anteriores.

Todas estas consultas se ejecutan en la base de datos local de VOS el cual tiene varios triples RDF. Es la misma base de datos que se utilizó en [Buil-Aranda y Rojas, 2020], el cual es una versión procesada y preparada para ser utilizada en VOS que está disponible en el sitio oficial de Wikidata⁷. Además, el *dataset* solo contiene consultas que utilizan el formulario SELECT.

Al ejecutar las consultas se tiene un problema, y es que la cantidad de consultas que tienen una latencia baja son mayoría. Esto genera un gran dilema pues, por un lado, las consultas que más latencia tienen son más interesantes de analizar y optimizar, pero por otro lado,

⁷https://www.wikidata.org/wiki/Wikidata:Database_download.

como se mencionó al comienzo de esta subsección, no es deseable un *dataset* extremadamente disperso. Asumiendo un máximo de latencia de 200 milisegundos, en la Figura 14 se observa a través de un histograma en escala logarítmica de la cantidad de consultas que se encuentran en distintos rangos de latencia, donde se observa que la mayoría se encuentran en rangos bajos. Si a estos le sumamos las subconsultas de latencia baja, tendríamos un *dataset* completamente desbalanceado.

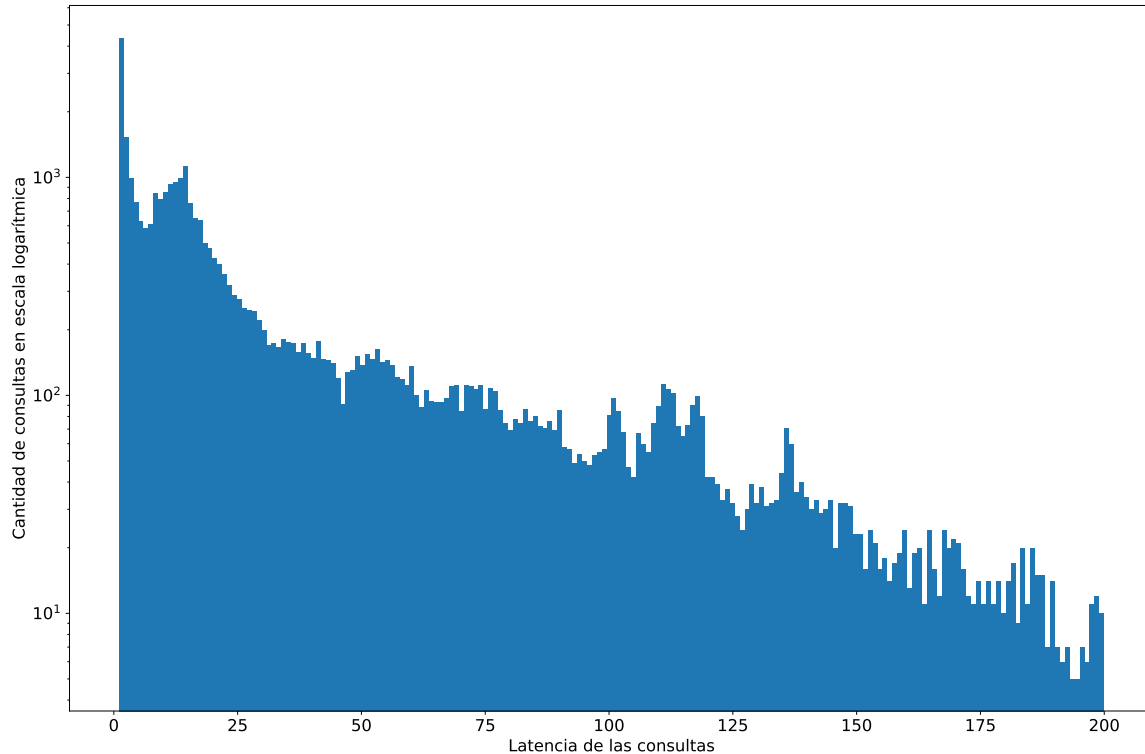


Figura 14: Histograma de latencia de las consultas sin procesar, hasta un máximo de 200 milisegundos de latencia.

Fuente: Elaboración propia.

Debido a eso, tras varios ensayos y errores, se tomó la decisión de realizar dos cortes de rango de latencia en milisegundos, uno de $[45, 150]$ para las consultas y otro de $[1, 150]$ para las subconsultas. Además, se realizó una limpieza de consultas que tuvieran algún error cuando pasaron por el extractor de características. Por ejemplo, algunas consultas no pudieron generar un árbol de ejecución pues su PROFILE generó error por *timeout*, por caracteres no permitidos, etc. Tras realizar el primer corte de rangos, antes de obtener las subconsultas extraemos el 25 % de esos datos con el fin de utilizarlos en los modelos de RL. Esto es, porque en esos modelos interesa optimizar consultas completas con latencias interesantes. Posteriormente, se extraen las subconsultas y se realiza el segundo corte de rangos. En resumen, una vez procesado y limpiado, el *dataset* contiene un total de 7955, y se divide de la siguiente manera:

- 75 % de los datos totales se utilizarán para entrenar el modelo de predicción de latencia. Incluye consultas y subconsultas. En total son 5995 datos, que se expanden a 10633 cuando se agregan las subconsultas, y estas se dividen de la siguiente manera:
 - 75 % se utilizará como datos de **entrenamiento**
 - 15 % se utilizará como datos de **validación**
 - 10 % se utilizará como datos de **prueba**
- 25 % de los datos totales se utilizarán para probar y entrenar políticas de modelos RL, correspondiendo a 2000 datos. Solo consultas completas, pues son estas las que realmente interesan optimizar.

3.2. Diseño de modelo DRL

El modelo propuesto como se ha mencionado anteriormente, consta de un *Deep Reinforcement Learning* que permita mejorar consultas SPARQL. Para esto, el modelo debe encontrar un plan de consultas mejor que el que emplea el gestor de base de datos VOS. Como los planes de consulta se están representando en árboles binarios, entonces puede concluirse que encontrar un mejor plan de consulta es equivalente a encontrar otro árbol binario con los mismos nodos. No obstante se debe tener una consideración especial: una vez escogido un nodo hoja que representa un triple, este debe unirse sí o sí con otro nodo hoja que represente un triple que **pertenezca al mismo BGP** (si existe). En otras palabras, si tenemos un BGP B con 3 triples, y otro B' con 2 triples, no puedo escoger un triple de B y luego otro de B' libremente. Esto se puede hacer si, y solo si, se incluyeron en el árbol todos los triples de B . A continuación se detallarán las partes **Deep Learning** y **Reinforcement Learning** del modelo propuesto.

3.2.1. Deep Learning

Se implementó un modelo de DL que permite predecir el tiempo de ejecución, o lo que es lo mismo, la latencia de la consulta. Para el entrenamiento se utilizaron tanto consultas con planes de ejecución completos P_f como subconsultas con planes de ejecución parciales $P_i \subset P_f$.

La **arquitectura de la red neuronal** es similar a la de [Marcus *et al.*, 2019] y [Casals *et al.*, 2020]. Sea E la experiencia de la red neuronal. Se tiene un conjunto de planes de ejecución $P_f \in E$ con una latencia conocida $L(P_f)$. Además, se tienen subplanes pertenecientes a planes de ejecución $P_i \subset P_f$, también con latencia conocida $L(P_i)$. Se debe entrenar el modelo M de tal manera que la latencia de los planes y subplanes se aproxime a la experiencia. Como un subplan parcial es, en esencia, un plan de consulta más

(o viceversa) bajo este contexto, el modelo se construye de la siguiente manera:

$$M(P_i, \theta^*) = \min \{C(P_i \mid P_i \subset P_f \vee P_f \in E)\}$$

Donde $C(P_i)$ es el costo de un subplan de ejecución $P_i \subset P_f$, y θ^* son los parámetros del modelo, los cuales deben ir ajustándose para que logren su objetivo, el cual es minimizar el error al estimar la latencia de las consultas y subconsultas. Para eso se utiliza una función de perdida $L2$ sobre el conjunto de entrenamiento $A = \{(x_n, y_n)\}_{n=1}^N$ (donde x_n son las características y y_n es el valor objetivo, es decir, la latencia), por tanto el *error de entrenamiento* es:

$$\theta^* = \operatorname{argmin}_{\theta} \operatorname{MSE}(A, \theta)$$

Donde MSE es el error cuadrático medio, y se define como:

$$\operatorname{MSE} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

La arquitectura de la red neuronal se diseñó para generar un buen *inductive bias* para la predicción de latencia de consultas. El diseño de la red codifica detalles de las posibles causas que producen la ejecución rápida o lenta de una consulta. Como se ha demostrado en [Marcus et al., 2019] y [Casals et al., 2020], la mayor influencia de las características a la hora de predecir a la latencia de consultas son los árboles que representan planes de ejecución. Por lo tanto, el modelo debe ser experto en el reconocimiento de los patrones presentes en estos subárboles, entonces, al igual que estos trabajos mencionados, se propone la técnica convolución de árboles, o **Tree Convolution**. Gráficamente, la arquitectura es muy similar a la Figura 10 presentada en el Capítulo 1, con la diferencia de que la codificación a nivel de consultas tiene dimensión 5. Como se ve en la figura mencionada:

- El modelo recibe en primer lugar las características a nivel de consultas pasándolas por varias capas completamente conectadas, o *Fully Connected Layers*.
- Posteriormente recibe las características a nivel de planes de ejecución y las concatena con el último vector producido por las *Fully Connected Layers* anteriores. Es decir, a cada nodo del árbol se le agregan las características a nivel de consultas previamente pasadas por las *Fully Connected Layers*. Esta técnica se le conoce como replicación espacial [Marcus et al., 2019], el cual sirve para combinar datos de tamaño fijo con datos de tamaño dinámico. Esto es, porque los datos a nivel de consulta siempre serán fijos respecto a la cantidad, para cualquier consulta, pero los datos a nivel de planes de ejecución, al ser representados como árboles siempre tendrán un tamaño distinto dependiendo la complejidad, el número de triples y el número de BGP's de la consulta.
- Este árbol aumentado, pasa por varias capas de convolución de árbol, o **Tree Convolution** [Mou et al., 2016], en donde cada capa devuelve un árbol con la misma estructura, pero con distintos tamaños en cada nodo.

- Posteriormente se aplica una agrupación dinámica, o **Dynamic Pooling** [Mou et al., 2016], el cual aplanar la estructura del árbol a un solo vector.
- Finalmente se pasa este vector aplanado por varias *Fully Connected Layers* hasta llegar a un valor único, el cual es la predicción de latencia obtenida.

Como **función de activación**, se utilizó *ReLU* para las capas de *Fully Connected Layers* y *Leaky-ReLU* para las capas de *Tree Convolution* (con *negative slope* 0, 01⁸). El tamaño de las capas se decidió en base a *tuning* de hiperparámetros (el cual se puede ver en el Capítulo 4), donde se determinó que se usarían tres capas *Fully Connected Layers* de tamaños (32,64,128) , y tres capas de *Tree Convolution* de tamaños (512,256,128).

A continuación, se explicarán ciertos detalles de la capa *Tree Convolution* y del *Dynamic Pooling*.

Tree Convolution

Tree Convolution, convolución de árbol, o simplemente *TreeConv*, es un tipo de red neuronal inspirado en las tradicionales *Convolutional Neural Networks* (también conocidas como CNN o *ConvNets*). La gran diferencia es que las *TreeConv*, en vez de recibir datos con estructura matricial, reciben datos con estructura de árbol. Se compone de varios filtros que pasan por todas las partes del árbol, capturando las relaciones entre nodos hijos y padres.

En particular para la memoria, la *TreeConv* ayuda en gran medida para capturar la influencia en la latencia de los predicados, los tipos de triples y el tipo de JOINS con los que se relaciona. Un filtro por ejemplo podría estimar las implicaciones del uso de un predicado con baja cardinalidad en el lado izquierdo o derecho de un JOIN, el costo asociado a ejecutar primero un tipo de triple determinado, etc. Por ejemplo, ejecutar primero un triple de tipo VAR_URI_URI o VAR_URI_LITERAL disminuye significativamente la cardinalidad de los resultados intermedios y finales, lo que puede traducirse en disminución en la latencia, mientras que un triple del tipo VAR_URI_VAR suele estar asociado a una alta cardinalidad, y por ende puede significar consultas de alta latencia.

En este trabajo, se utilizó la versión de [Casals et al., 2020] de *TreeConv*⁹ (con pequeñas modificaciones), el cual es una versión similar a [Marcus et al., 2019] pero se utilizó la versión simplificada a árboles binarios de la original¹⁰. Sin embargo, esta versión de *TreeConv* está perfectamente adaptada para leer y utilizar árboles que representen planes de ejecución de consultas SPARQL con relativa facilidad. Corresponde a una convolución 1D, similar a *ConvNet1D*. La secuencia se genera y recorre el árbol en preorden, teniendo en cuenta tanto la raíz los hijos izquierdo y derecho. Por tanto, cada paso será de "(Padre-hijo izquierdo-hijo

⁸Hiperparámetro propio de la función de activación *LeakyReLU*. Más información en: <https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html>.

⁹Código de [Casals et al., 2020], aquí se puede encontrar el código de *Tree Convolution* adaptado a árboles binarios SPARQL: https://github.com/dacasals/sparql_neo/.

¹⁰*Tree Convolution* utilizado en Neo, simplificado para árboles binarios: <https://github.com/RyanMarcus/TreeConvolution>.

derecho)”. Las secuencias son variables, ya que los árboles tienen distinto tamaño dependiendo del plan de ejecución. Para solucionar esto, se agregan nodos vacíos hasta alcanzar el árbol de mayor profundidad del *dataset*.

Cada filtro contiene tres vectores de ponderaciones correspondientes a la raíz y a los dos hijos, los cuales llamaremos $\mathbf{e}_p, \mathbf{e}_l, \mathbf{e}_r$, formando una suerte de filtro en forma de “triángulo”. Cada filtro se aplica a cada triángulo local $\mathbf{x}_p, \mathbf{x}_l, \mathbf{x}_r$, tal que $\mathbf{x}_p$ es un vector que puede representar un nodo intermedio. En ese caso, $\mathbf{x}_l$ y $\mathbf{x}_r$ son vectores que representan a los dos hijos. No obstante $\mathbf{x}_p$ podría ser también un vector que representa un nodo hoja. En ese caso los vectores hijos serán vectores . En cualquier caso, se le denomina a este triángulo un **nodo árbol** x'_p . Expresado matemáticamente, el árbol sería la siguiente transformación no lineal:

$$x'_p = \sigma (\mathbf{e}_p \cdot \mathbf{x}_p + \mathbf{e}_l \cdot \mathbf{x}_l + \mathbf{e}_r \cdot \mathbf{x}_r)$$

Donde ϕ es una función de activación, que en nuestro caso será *LeakyReLU*, y $(\cdot)$ es un producto escalar entre vectores. De esta forma, el filtro es capaz de obtener información no solo de los nodos individuales, si no que también información de la relación entre nodos.

Como la salida de una *TreeConv* es otro árbol, se pueden aplicar varias capas de filtro de *TreeConv*. La primera capa accedera al árbol de plan de ejecución aumentado con la información a nivel de consulta. La segunda capa de filtros se aplicará ya a un árbol con información procesada por la primera capa, derivada de un nodo n en el árbol aumentado original, los hijos n y los nietos n , es decir, cada capa de convolución tiene un campo receptivo más grande que el anterior. Por lo tanto, la primera capa *TreeConv* aprende características simples, mientras que la última capa de *TreeConv* aprende características complejas. En la Figura 15 se muestra como *TreeConv* actúa sobre dos planes de ejecución que difieren en el operador JOIN. Se puede observar en la figura que el tipo de JOIN está codificado en las dos primeras entradas del vector de características en cada nodo. Un filtro de *TreeConv*, compuesto por tres vectores de peso con $\{1, -1\}$ en las dos primeras posiciones y ceros para el resto, servirá como “detector” para planes de consulta con dos combinaciones de JOINS secuenciales. Obsérvese como el nodo raíz del plan con dos combinaciones de JOINS secuenciales recibe una salida de 2 de este filtro, mientras que el nodo raíz del plan con una combinación LEFT JOIN en la parte superior de un JOIN recibe una salida de 0. Las *TreeConv* posteriores pueden usar esta información para formar detectores más complejos.

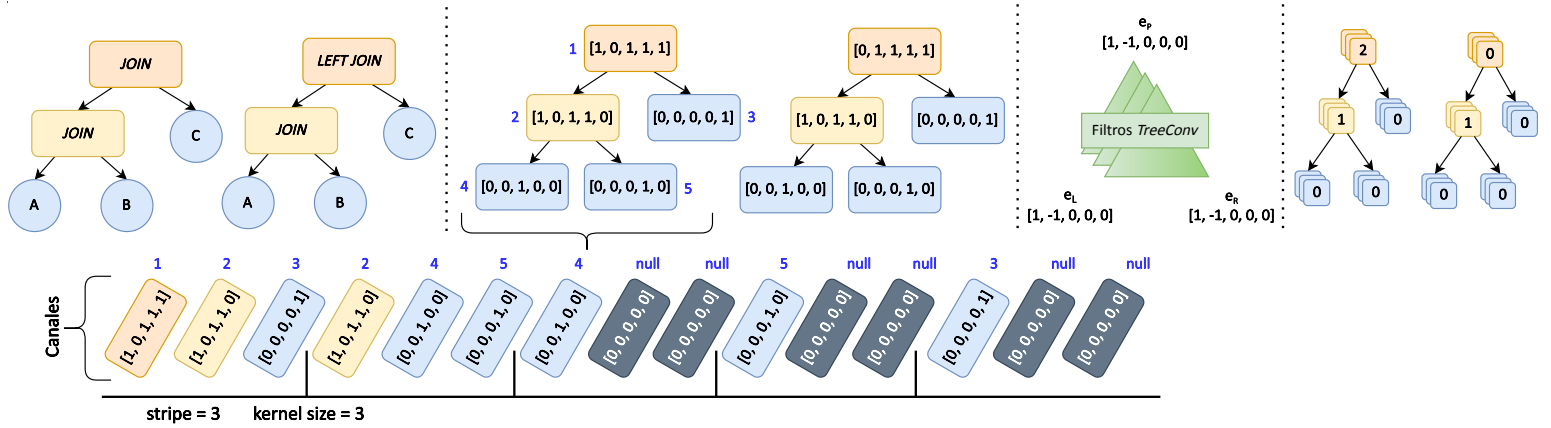


Figura 15: Ejemplo de Tree Convolution.

Fuente: [Casals et al., 2020].

Dynamic Pooling

Posterior a las capas de *TreeConv*, viene una capa de *pooling* dinámico, o *Dynamic Pooling*. Esta tiene como objetivo, aplanar toda la información de un árbol procesado ya por las *TreeConvs*, a un vector fijo grande, que posteriormente pasará por capas de *Fully Connected Layers* disminuyendo la dimensión de este vector hasta un valor fijo el cual será la latencia que se buscaba predecir. El tamaño del árbol que aplanar esta capa es exactamente igual que el original, pues como se mencionó anteriormente las *TreeConv* conservan la estructura del árbol de planes de ejecución. Similar a las capas de *pooling* presentes en las redes neuronales de tipo *ConvNet* [Socher et al., 2011], la capa de *pooling* dinámico toma el valor más influyente en cada dimensión de las características que son detectadas por la *TreeConv*. Cada valor que toma va agrupándolas en un vector que será apto para pasar por las *Fully Connected Layers* siguientes. Finalmente, la red puede entrenarse en un modelo supervisado utilizando *backpropagation*.

3.2.2. Reinforcement Learning

El aprendizaje por refuerzo, *Reinforcement Learning* o simplemente RL como se mencionó en secciones anteriores, es paradigma de ML en donde, en pocas palabras, consta de “un agente que va aprendiendo de su entorno”. Como RL está basado en MDP, lo primero que se debe hacer es precisamente replantear el problema como un **problema MDP**. En este caso, replantear a MDP la optimización de consultas SPARQL. Se debe definir entonces elementos como el agente *learner*, el entorno, las acciones, recompensas, entre otros, pero también se debe decidir que algoritmos de función de valor se utilizarán. Además, se deben *tunear* los hiperparámetros correspondientes para lograr el mejor resultado posible. Para lograr desarrollar un modelo de RL, nos apoyaremos en la caracterización de consultas realizada, principalmente en los árboles de ejecución, así como también en el modelo DL de predicción de

latencia.

Modelo de RL general

Para poder construir el modelo de RL, primero se hacen las siguientes definiciones:

- **Entorno:** El entorno es el total de nodos que contiene el árbol. Es decir, todos los árboles y subárboles.
- **Agente:** En el caso de este problema, el agente *learner* es un elemento abstracto, pues no es fácil identificar un agente propiamente tal, ya que no se trata de un problema fijo, si no, más bien uno dinámico donde el tamaño del árbol de ejecución es variable. Tampoco es un problema que se pueda utilizar un elemento físico o virtual reconocible, como podría ser un robot, un jugador manejado por inteligencia artificial, etc. Para este problema, se puede pensar como un agente, a aquel que define cuál es la siguiente subconsulta que se ejecutará a medida que se va construyendo el árbol.
- **Estados:** Serían todos los posibles árboles parciales existentes en un árbol de ejecución, es decir, todos los subplanes en un plan de ejecución. El estado inicial sería el momento previo a comenzar a construir el árbol, es decir, una estructura vacía. El estado siguiente sería el resultante tras escoger cualquier nodo que represente una subconsulta, el estado subsiguiente sería el resultante tras escoger ya dos subconsultas más el nodo intermedio que los une, y así sucesivamente. Es decir, cada vez que se agrega un nuevo nodo al árbol que se está construyendo, el agente se mueve a un nuevo estado. El estado final es cuando el árbol se a completado en su totalidad utilizando todas las subconsultas. En otras palabras, cada subárbol es un estado, y el árbol completo, donde ya se escogieron todas las acciones, es el estado terminal.
- **Acciones:** Las acciones son todos los posibles nodos hojas que se pueden escoger a partir del estado presente, es decir, todas las subconsultas de un plan de consultas. Por lo tanto, se debe utilizar una **heurística**, en donde las acciones posibles de un determinado nodo hoja deben si o si respetar el BGP presente. Es decir, si un nodo hoja representa una subconsulta que se ejecuta en un BGP B , la siguiente acción debe ser otra subconsulta presente en el BGP B si y solo si la cantidad de subconsultas disponibles en B no es nula. En otro caso puede escogerse entonces una subconsulta presente en otro BGP B' . Además, esta heurística identifica que la siguiente acción no puede ser la misma acción usada para llegar al estado actual, es decir, no se puede repetir una acción. Esta heurística la denominamos **heurística de acciones disponibles**. El número de acciones escogidas tras llegar a un estado terminal serán todas las acciones posibles. Este dato sirve para poder construir la política inicial.
- **Recompensa:** La recompensa se define dependiendo el algoritmo de función de valor que se utilizará. En caso de que el algoritmo sea el propuesto por Neo, la recompensa será 0 si el estado no es terminal y el negativo de la latencia producida por el árbol en caso de ser estado terminal. Para los otros algoritmos siempre la recompensa será el negativo de la latencia del árbol parcial producido por el estado de ese momento.

- **Política:** Será una matriz $Q_{S \times A}$, donde S es el largo menos uno del estado terminal y A el largo menos uno del conjunto de acciones posibles.

A continuación se describirá como funciona el flujo general:

1. En primer lugar se definen los hiperparámetros (tasa de aprendizaje, factor de descuento, etc.) y la cantidad de iteraciones.
2. Se escoge una consulta que se desea optimizar, incluyendo las características a nivel de consulta y las características a nivel de plan de ejecución (el árbol de ejecución) . También se recibe un diccionario con los BGPs y las consultas que ejecutan esos BGPs, ordenados según como se ejecutaron en el plan de consultas. De esta forma se tiene el total de subconsultas y los BGP a cuál pertenecen.
3. Teniendo ya las subconsultas, se crea una lista con todas las acciones posibles.
4. Se crea una matriz de 0 de tamaño $S \times A$ el cuál representará la política inicial Q . Nótese que el valor de S ya se tiene pues sabemos que la cantidad de acciones en un estado terminal es igual a la cantidad de acciones posibles. En realidad, se podría decir que en este caso particular la matriz es $S \times S$ o $A \times A$, pero por comodidad en la sintaxis se mantendrá la notación estándar.
5. Se hace un ciclo de tipo *for* para cada iteración.
6. Se crea una lista vacía que representa el estado inicial s_0 .
7. Se escoge alguna acción inicial a_0 . Para esto se utiliza un algoritmo ε -greedy, esto es, dado un valor ε se escoge un valor aleatorio $v \in [0, 1)$. Si v es mayor que ε , entonces la siguiente acción será escogida aleatoriamente respetando la heurística de acciones disponibles. Caso contrario, se elige de acuerdo al valor *argmax* de la política del estado próximo s_0 , también respetando la heurística de acciones disponibles (en otras palabras, se escoge la mejor acción posible de forma "greedy"). A mayor valor de ε , menor exploración, y viceversa.
8. Se pasa al estado inicial s_1 una vez escogida la acción inicial a_0 .
9. Comienza un ciclo del tipo *while*, que se extiende hasta que el estado sea terminal.
10. Se escoge una nueva acción a_{i+1} distinta de a_i , y respetando la heurística de estados disponibles, a través del algoritmo ε -greedy ya descrito anteriormente.
11. Se pasa desde un estado s_i a otro estado s_{i+1}
12. Se guarda la recompensa R .
13. Se actualiza la política en la posición $Q[s_i, a_i]$ de acuerdo a algún algoritmo de función de valor.

La Figura 16 es un pseudocódigo general de como se realiza todo el proceso para optimizar una consulta a través de un modelo RL, en donde se puede apreciar de mejor manera el flujo. La Figura 17 es un diagrama donde se ve como un árbol pasa de un estado a otro

a través del flujo, donde en cada estado tiene una cantidad de acciones disponibles a elegir. Las acciones como mencionamos, representa una subconsulta que se va a escoger. No obstante como el orden es importante (por eso la existencia de la heurística de acciones disponibles), en la práctica cuando el agente escoja una acción, el agente obtendrá mucha más información, pues además del predicado y el tipo de triple, recibirá información como el BGP en que se encuentra la subconsulta (para no entremezclar BGP), si está dentro de un operador *OPTIONAL* (para elegir el tipo de *JOIN* al momento de unir BGPs), entre otros. Con estos datos se puede crear una función que represente la heurística de acciones disponibles. Finalmente, la Figura 18 es un diagrama que representa como una política pasa por varias iteraciones hasta una política final y como esta se expresa en un árbol de ejecución de consultas (además de remarcar en distintos colores los BGPs, y expresar con flechas el orden en que fueron escogidos).

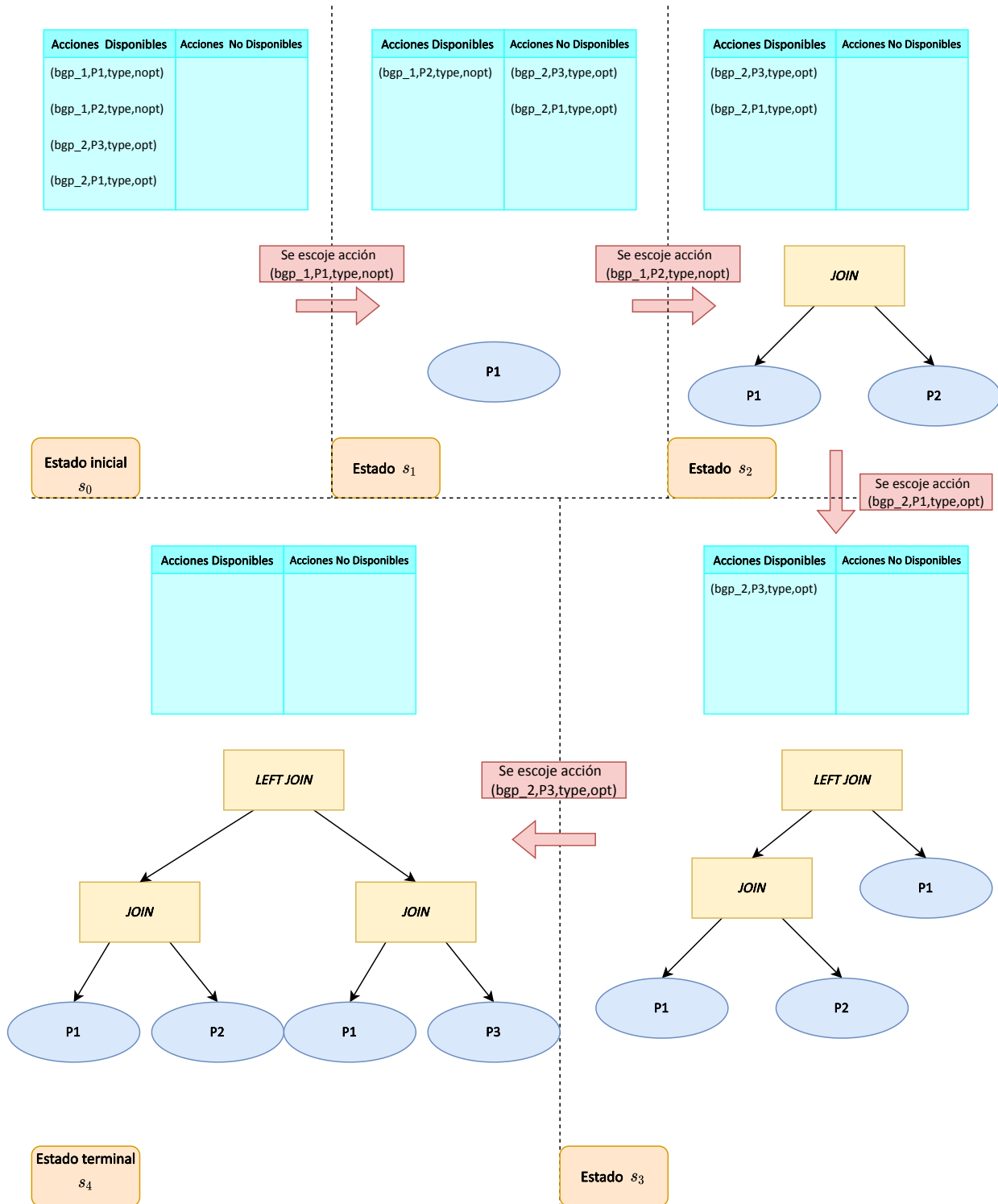


Figura 17: Ejemplo de flujo del proceso de generar un nuevo árbol de ejecución de consultas en una consulta genérica con dos BGPs (uno de ellos del tipo OPTIONAL), cuatro subconsultas, y tres predicados. Este proceso se realiza en cada iteración.

Fuente: Elaboración propia.

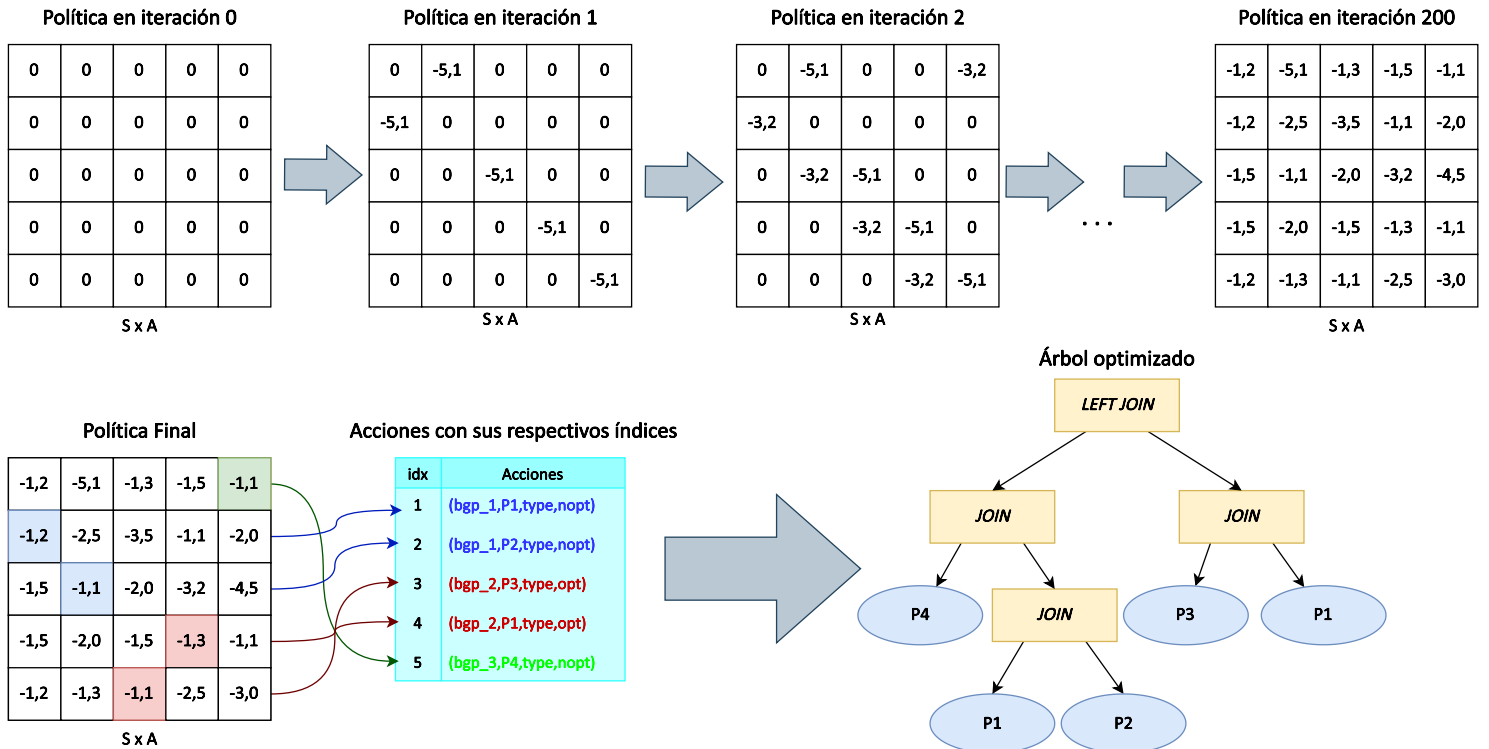


Figura 18: Ejemplo de como varía una política utilizando el algoritmo Neo, junto con el árbol final que generaría para una consulta genérica con tres BGPs (uno de ellos del tipo OPTIONAL), cinco subconsultas, y cuatro predicados.

Fuente: Elaboración propia.

Algoritmos de funciones de valor utilizadas

■ Función de valor de Neo

Cuando se habla de la función de valor de Neo, se hace referencia a la que se utilizó en [Marcus *et al.*, 2019], el cual no es ningún algoritmo de RL tradicional, sino más bien, uno pensado para este problema en particular. Se diferencia de los otros, ya que este asigna valores en la política solo en el estado terminal, incluyendo valores ya procesados. Es decir, la recompensa en este algoritmo es discreta y se define como:

$$R(s_i, a_i) = \begin{cases} 0 & : i \Rightarrow \text{Estado No Terminal} \\ -pred(new_tree, query_values) & : i \Rightarrow \text{Estado Terminal} \end{cases}$$

Donde s_i y a_i son el estado terminal y la acción tomada en el paso i terminal, new_tree es el árbol construido en i , $query_values$ son las características a nivel de consultas de ese árbol, y $pred$ es una función que llama al modelo DL con los parámetros ya mencionados, obteniendo la latencia.

La política, como se dijo al comienzo, se actualiza solo en un estado terminal, actualizando todos los valores anteriores, similar a la Figura 11. En el Anexo puede encontrar un pseudocódigo específico del algoritmo Neo completo.

■ Q-Learning, Sarsa y Expected Sarsa

Los algoritmos *Q-Learning*, *Sarsa* y *Expected Sarsa* son clásicos cuando se habla de RL. Los tres son algoritmos basados en el concepto de *Temporal Difference* (o simplemente TD), un enfoque para aprender a predecir y/u optimizar una cantidad que depende de los valores futuros. Un algoritmo *off-policy* evalúa o mejora una política diferente de la utilizada para generar los datos. Usualmente, es más apropiado cuando el agente no explora mucho. Por su parte un algoritmo *on-policy* intentan evaluar o mejorar la política como tal que se utiliza para tomar decisiones, siendo usualmente más apropiado cuando el agente explora bastante. *Q-Learning* es un algoritmo *off-policy*, mientras que *Sarsa* es un algoritmo *on-policy*. Por su parte *Expected Sarsa* será *on-policy* u *off-policy* dependiendo de la situación. Como no se ha usado demasiado RL en la literatura para la optimización de consultas SPARQL, se utilizaron y compararon los tres algoritmos (además de Neo). Para más información teórica de estos algoritmos puede consultar [Sutton y Barto, 2018]. Para el caso de este proyecto se utilizó una versión levemente modificada de los tres.

La recompensa en los tres algoritmos es recibida de la siguiente manera:

$$R(s_i, a_i) = -pred(partial_tree, partial_query)$$

Donde s_i y a_i son el presente estado y la acción tomada en el paso i , $partial_tree$ es el subárbol parcial construido hasta el paso i , $partial_query$ son las características a nivel de consultas de ese árbol parcial, y $pred$ es una función que llama al modelo DL con los parámetros ya mencionados, obteniendo la latencia.

Por su parte actualizar la política dependerá del algoritmo que se este utilizando. Para **Q-Learning** la función de valor es:

$$Q(s_i, a_i)_{ql} = Q(s_i, a_i)_{ql} + \alpha(R_{i+1} + \gamma \max_a Q(s_{i+1}, a)_{ql} - Q(s_i, a_i)_{ql})$$

Para **Sarsa** la función de valor es:

$$Q(s_i, a_i)_{sar} = Q(s_i, a_i)_{sar} + \alpha(R_{i+1} + \gamma Q(s_{i+1}, a_{i+1})_{sar} - Q(s_i, a_i)_{sar})$$

Finalmente, para **Expected Sarsa** la función de valor es:

$$Q(s_i, a_i)_{exsar} = Q(s_i, a_i)_{exsar} + \alpha(R_{i+1} + \gamma \sum_a \pi(a|s_{i+1}) Q(s_{i+1}, a)_{exsar} - Q(s_i, a_i)_{exsar})$$

Como la acción se elige con ε -greedy con la heurística de estados disponibles, se define π como:

$$\pi(a|s_i) = \begin{cases} \frac{\varepsilon}{A_{disponibles}} & : a \Rightarrow \text{acción es greedy} \\ 1 - \varepsilon + \frac{\varepsilon}{\text{Número de acciones greedy}} & : a \Rightarrow \text{acción no es greedy} \end{cases}$$

Luego, Q_{ql} , Q_{sar} y Q_{exsar} son la funciones de valor de **Q-Learning**, **Sarsa** y **Expected Sarsa** respectivamente, s_i y a_i son el estado y la acción tomados en el momento i , a es cualquier acción tal que $\forall a \in A_{disponibles}$ lo cual quiere decir que la acción está respetando la heurística de acciones disponibles. Esto quiere decir por ejemplo, que en el algoritmo de **Q-Learning** o de **Expected Sarsa** no se calcula el máximo de la fila que represente ese estado en la política, sino que el máximo de algunas columnas de esa fila, es decir, de las acciones disponibles que pueden escogerse en esa política. Esto se hace, ya que no tiene sentido contaminar la política con valores imposibles para el estado-acción que se está evaluando.

Continuando con otros símbolos en las ecuaciones, α es la tasa de aprendizaje, γ es el factor de descuento de aprendizaje, R_i es la recompensa tomada en el paso i , y ε es el valor para el algoritmo de ε -greedy al escoger una acción. En el Anexo puede encontrar un pseudocódigo específico de cada algoritmo TD.

Valores como α , γ y ε , son hiperparámetros de entrada y deben *tunearse*.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

En este capítulo se explicarán los distintos procesos, experimento y comparaciones con el fin de comprobar si los modelos cumplen, en mayor o menor medida, tanto el objetivo principal como algunos objetivos específicos que se plantearon en la memoria. Se evaluó tanto la parte DL, es decir, el modelo de **predicción de latencia**, como la parte RL, lo que hace referencia a la **optimización de consultas** como tal. Se explicarán las métricas de evaluación para el modelo de predicción. También, se detallará el **tuning de hiperparámetros** tanto para la predicción como para la optimización de latencia. Por último, se hizo un **análisis con los mejores modelos** obtenidos en ambos casos.

4.1. Predicción de latencia

4.1.1. Métricas de Evaluación

Se debe tener en cuenta que la predicción de latencia es un problema de **regresión**, es decir, se busca predecir un valor continuo de acuerdo a los valores de los pesos obtenidos por el **modelo**. Además, el modelo se entrenó bajo un paradigma **supervisado**. Por tanto, las métricas escogidas deben ser aquellas utilizadas para medir este tipo de modelos. Las métricas escogidas fueron:

- **Raíz del error cuadrático medio (RMSE):** El error cuadrático medio (MSE) es la métrica que posiblemente sea la más utilizada para comparar estimadores en regresión, sobre todo si el modelo es supervisado. También se utiliza bastante como función de pérdida en modelos con redes neuronales. Se calcula, como se mencionó anteriormente, de la siguiente manera:

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

La **Raíz cuadrada del error cuadrático medio (RMSE)** es una variante del MSE utilizado como métrica por [Hasan y Gandon, 2014], el cual tiende a penalizar errores muy grandes. Se calcula de la siguiente manera:

$$RMSE = \sqrt{\frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2}$$

- **Error absoluto medio (MAE):** Se define como la distancia vertical promedio entre cada uno de los puntos y la recta identidad, y es otra métrica de las más utilizadas en

problemas de regresión con modelos supervisados. Se calcula de la siguiente manera:

$$MAE = \frac{1}{m} \sum_{i=1}^m |y_i - \hat{y}_i|$$

4.1.2. *Tuning* de hiperparámetros

Se realizaron varias pruebas de *tuneo* de hiperparámetros. Todos tienen una base fija, donde se usa la función *ReLU* como función de activación para las *Fully Connected Layers*, la función *LeakyReLU* (con valor *negative slope* 0, 01) como función de activación en las capas *TreeConv*, 180 épocas de iteración y el mismo *dataset* mencionado en el Capítulo 3.

***Tuning* en de función de optimización**

La función de optimización, se utiliza, en resumidas cuentas, para medir los pros y contra de un modelo. En este caso, se entrenaron tres modelos utilizando la base mencionada, una tasa de aprendizaje de 0,00015, tres capas de *Fully Connected Layer* de tamaños (32,64,128), y tres capas de *TreeConv* de tamaños (512,256,128). En cada modelo se probaron tres funciones de optimización: *Adam*, *AdaGrad* y *SGD*. El objetivo es comparar estas tres funciones de optimización que suelen ser típicas en los modelos de regresión supervisados. En la Figura 19 se ven los desempeños de cada uno en términos de RMSE y MAE. Se observa que *SGD* es el de peor resultado al converger más lento que las otras funciones. Por otra parte, se observa que *Adam* tiene un desempeño levemente mejor que *AdaGrad*. Por lo tanto, se decidió usar el optimizador *Adam*.

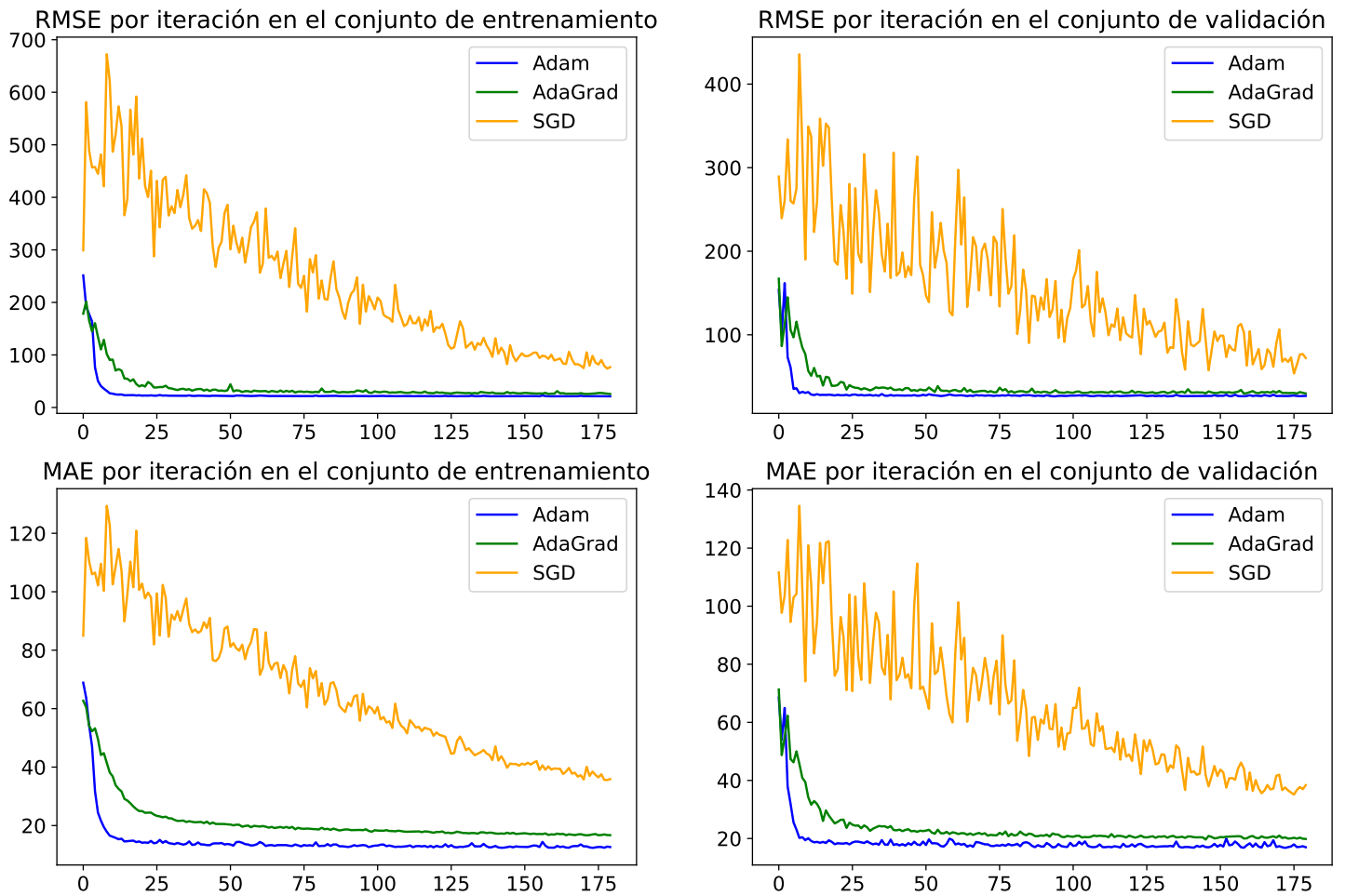


Figura 19: RMSE y MAE por iteración, en modelos similares con distintos optimizadores.
Fuente: Elaboración propia.

Tuning en de tasa de aprendizaje

En ML y la estadística en general, la tasa de aprendizaje es un hiperparámetro de ajuste en un algoritmo de optimización que determina el tamaño del paso en cada iteración mientras se mueve hacia un mínimo de una función de pérdida[Murphy, 2012]. El objetivo al *tunear* este hiperparámetro, es analizar en cuál orden de magnitud se consigue un modelo más estable. Se entrenaron cuatro modelos utilizando la base mencionada, tres capas de *Fully Connected Layer* de tamaños (32,64,128), y tres capas de *TreeConv* de tamaños (512,256,128) y la función de optimización *Adam*. En cada modelo se probaron cuatro tasas de aprendizaje con distintos órdenes de magnitud: 0,0000015, 0,000015, 0,00015 y 0,0015. En la Figura 20 se graficaron los desempeños obtenidos en términos de RMSE y MAE. Los valores más pequeños, con orden de magnitud igual o menores a 10^{-5} (0,0000015 y 0,000015), mostraron un desempeño peor comparado a los otros, al tener una convergencia más lenta, además de llegar a valores peores que las otras opciones. El mayor valor de todos, 0,0015, si bien fue el que convergió más rápido, también mostró mayor inestabilidad, sobre todo en el conjunto de validación. Tanto en RMSE como en el MAE se observan varias crestas en la curva, algunas con valores muy altos. Además, el resultado final no fue muy distinto a 0,00015. Por lo tanto, se decidió utilizar 0,00015 como tasa de aprendizaje.

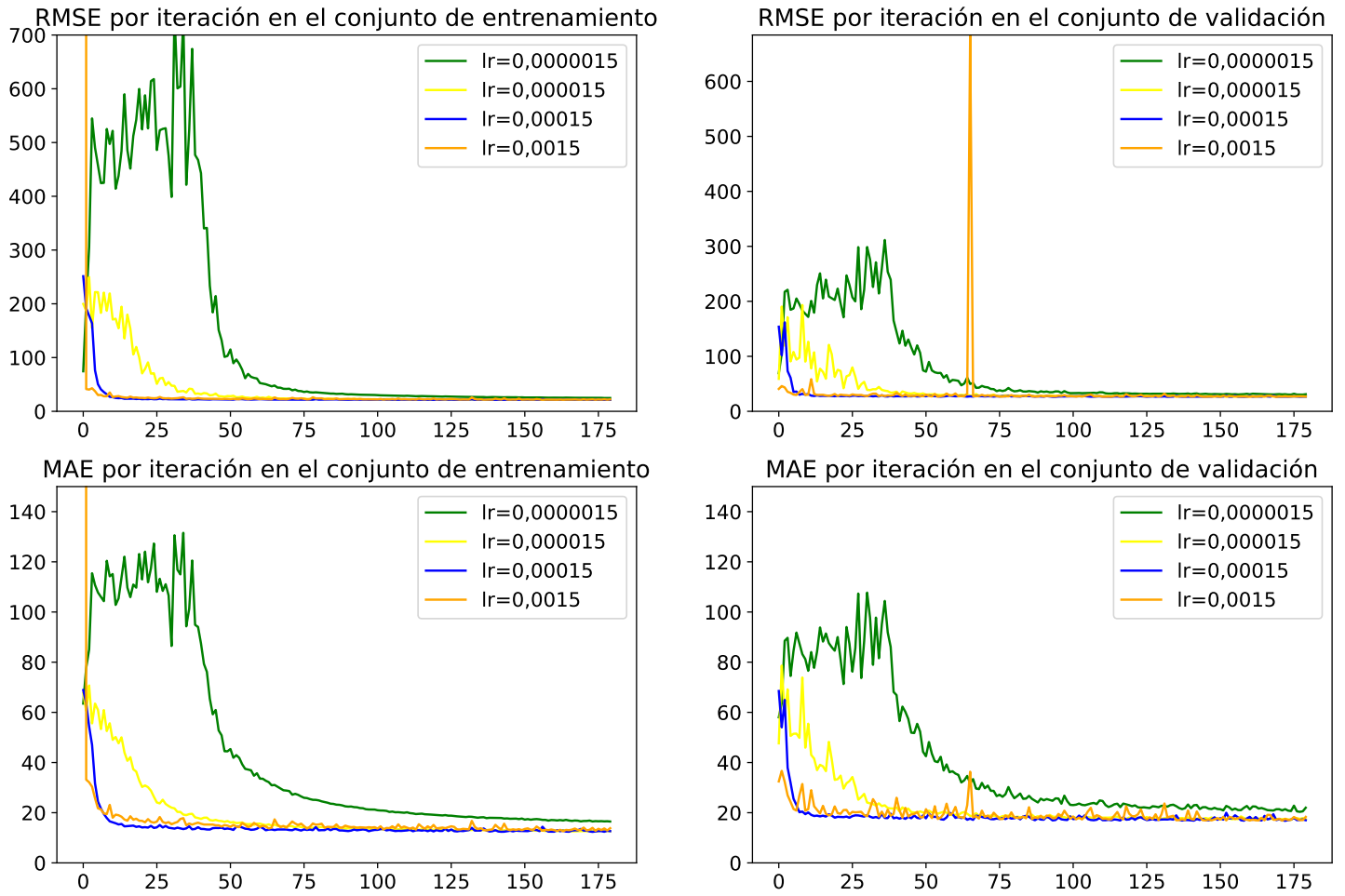


Figura 20: RMSE y MAE por iteración, en modelos similares con distintas tasas de aprendizaje.

Fuente: Elaboración propia.

Tuning en de capas ocultas

Utilizar distintas cantidades de capas, y variar los tamaños, puede añadir o quitar complejidad al modelo. Un modelo más complejo no necesariamente es mejor, por lo cual es importante encontrar la cantidad de capas que logre los mejores resultados. Se entrenaron seis modelos usando la base mencionada, la función de optimización *Adam* y 0,00015 de tasa de aprendizaje. En cada modelo se probaron seis combinaciones de capas ocultas de distintas complejidades y tamaño:

- **L1:** Tres capas de *Fully Conneted Layer* de tamaños (16,32,128), y una capa de *TreeConv* de tamaño (128).
- **L2:** Tres capas de *Fully Conneted Layer* de tamaños (32,64,128), y dos capas de *TreeConv* de tamaños (256,128).
- **L3:** Dos capas de *Fully Conneted Layer* de tamaños (16,32), y tres capas de *TreeConv* de tamaños (128,64,32).
- **L4:** Tres capas de *Fully Conneted Layer* de tamaños (16,32,64), y tres capas de *TreeConv* de tamaños (256,128,64).
- **L5:** Tres capas de *Fully Conneted Layer* de tamaños (32,64,128), y tres capas de *TreeConv* de tamaños (512,256,128).
- **L6:** Tres capas de *Fully Conneted Layer* de tamaños (32,128,128), y cuatro capas de *TreeConv* con tamaños (1024,512,256,128).

En todas las combinaciones, para las capas de *Fully Conneted Layer* se utilizó la función de activación *ReLU* y para las capas de *TreeConv* se utilizó la función de activación *LeakyReLU* con *negative slope* 0,01. En la Figura 21 se observan los desempeños obtenidos en términos de RMSE y MAE de cada combinación de capa. Como no se aprecian diferencia significativas, se decide seguir utilizando L5, el estándar utilizado hasta el momento.

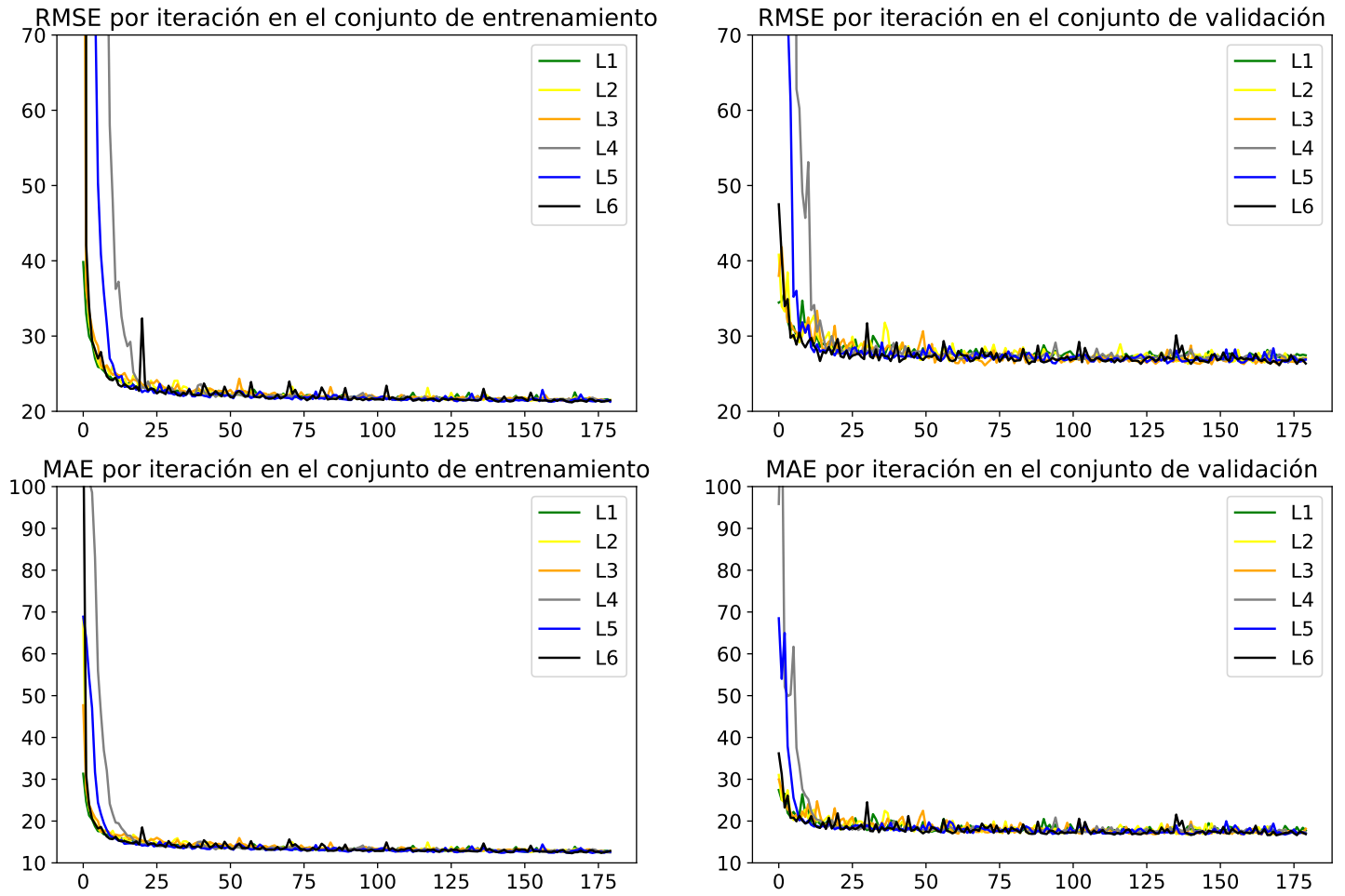


Figura 21: RMSE y MAE por iteración, en modelos similares con distintos tamaños de capas ocultas.

Fuente: Elaboración propia.

4.1.3. Análisis al mejor modelo de predicción de latencia obtenido

Tras el *tuneo* realizado, el mejor modelo obtenido tiene los siguientes valores de hiperparámetros:

- 180 iteraciones.
- Función de optimización *Adam*.
- Tasa de aprendizaje 0,00015.
- Tres capas de *Fully Connected Layer* de tamaños (32,64,128), con función de activación *ReLU* y tres capas de *TreeConv* de tamaños (512,256,128), con función de *LeakyReLU* (con *negative slope* 0,01).

La arquitectura de la red neuronal es similar a la de la Figura 10 en el Capítulo 1, con la diferencia de que en este caso la codificación de entrada a nivel de consulta no es 64, sino que es 5, ya que se utilizan menos características a nivel de consulta. En la Figura 22 se hicieron gráficos de puntos donde se comparan los valores de latencia de las consultas reales versus las predicciones de latencia del modelo, para los conjuntos de entrenamiento, validación y prueba. Cada punto representa una consulta. El eje horizontal están los valores predichos por el modelo mientras que el eje vertical es el tiempo de ejecución real, ambos en milisegundos. Para una consulta cualquiera, sea y el valor de latencia predicha por el modelo e $\hat{y}$ el valor real de latencia de esa consulta. Se define la diferencia porcentual absoluta $\%dpa$ de la consulta como:

$$\%dpa = \left| 100 - \frac{y}{\hat{y}} \cdot 100 \right|$$

Las predicciones se clasifican en la figura mediante colores, que representan lo siguiente:

- Si $\%dpa < 15 \%$, la predicción se considera buena (puntos verdes).
- Si $15 \leq \%dpa < 30 \%$, la predicción se considera aceptable (puntos azules).
- Si $\%dpa \geq 30 \%$, la predicción se considera mala (puntos rojos).

Se observa que la mayoría de los valores se encuentran en una situación buena y/o aceptable. No obstante, también se observa que existen varias predicciones malas, donde la mayoría son las de más de baja latencia. Se puede concluir que el modelo predice de mejor manera consultas de alta latencia.

En la Figura 23 se graficaron los valores de RMSE y MAE para cada conjunto de datos. Se observa el valor de RMSE en el conjunto de entrenamiento es mayor que en el conjunto de validación. El RMSE de entrenamiento es cercano a 21, mientras que en el de validación

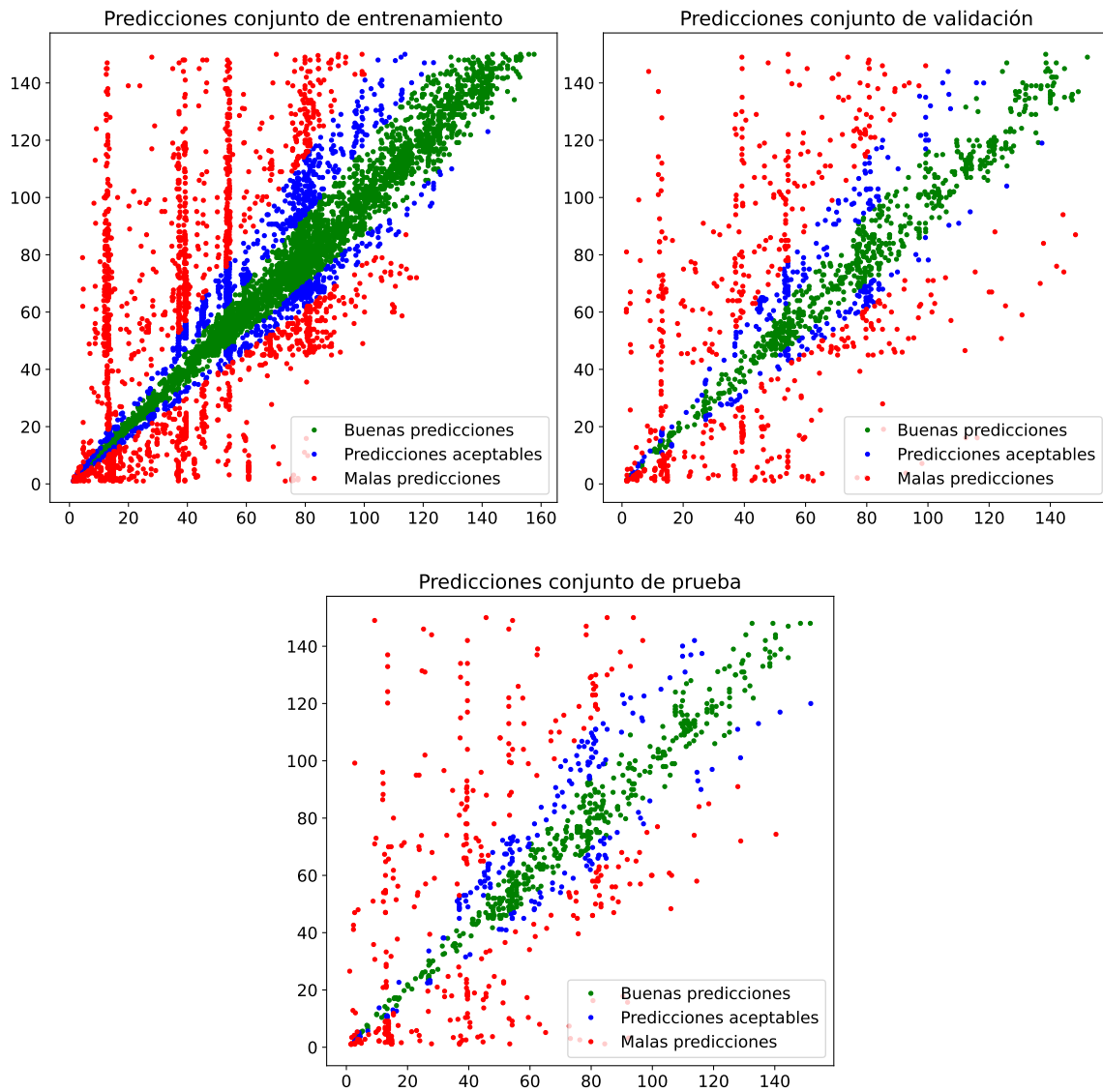


Figura 22: Comparación de valores de latencia reales versus las predicciones del modelo final.

Fuente: Elaboración propia.

es cercano a 27. Caso similar ocurre en el MAE, donde el valor es cercano a 15 el conjunto de entrenamiento, mientras que en el conjunto de validación es cercano a 20. Si bien estos valores obtenidos no son del todo malos, si podrían mejorarse. Además, el hecho de que estos valores en el conjunto de validación sean levemente superiores, indica cierto grado de *overfitting*. De todas maneras, al ser el mejor modelo obtenido, se decide seguir adelante.

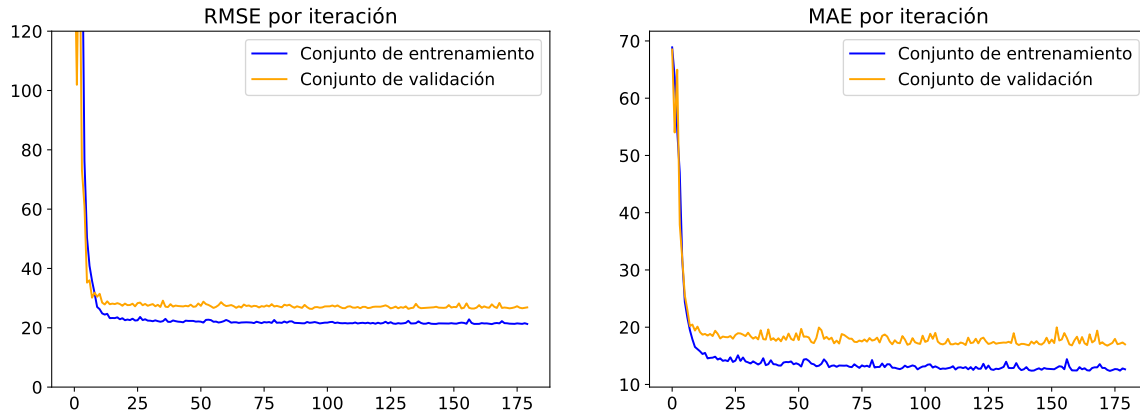


Figura 23: RMSE y MAE del modelo final.

Fuente: Elaboración propia.

4.2. Optimización de consultas

Las métricas de evaluación en este caso son bastante sencillas: básicamente se evalúa cada algoritmo en cada consulta del *dataset*, y luego se analizó la convergencia de aprendizaje y el promedio de latencia obtenido. Para los modelos finales, además se evaluó la cantidad de errores y aciertos, y se comparó el promedio de latencia con los valores reales de VOS. Recordar que el *dataset*, en este caso, solo contiene consultas, pues son estas las que interesan optimizar.

4.2.1. Tuning de hiperparámetros

Al igual que con el modelo de predicción de latencia, se hicieron varias pruebas de *tuneo* de hiperparámetros. En particular, se quiso observar como afecta el grado de exploración en cada modelo, modificando el valor del ε , para el algoritmo ε -greedy. Además, se probaron distintos valores de tasa de aprendizaje α y de factor de descuento γ . Todos tuvieron como base 200 épocas de iteración, el mismo *dataset* destinado a RL descrito en el Capítulo 3 y el mismo modelo de predicción de latencia, el cual fue el mejor obtenido detallado en la subsección anterior.

Tuning de ε -greedy

El valor del ε como mencionamos antes, hace referencia a la exploración a través del algoritmo ε -greedy. El valor de ε debe ser entre 0 y 1, donde si es cercano a 0 entonces la exploración es alta, mientras que un valor de exploración cercano a 1, indica que se desea enfocar el modelo a la explotación. Una gran ventaja de ε -greedy es que, en contraste a usar un simple elección greedy, es que a medida que aumenta el número de iteraciones, cada acción será elegida un número infinito de veces, existiendo probabilidades más altas de que la política converja lo mejor posible, logrando por ende, la mejor optimización posible [Sutton y Barto, 2018]. Se entrenaron 16 modelos, utilizando la base mencionada, 0,9 como valor de tasa de aprendizaje α y 0,999 como valor factor de descuento γ . Se utilizaron los cuatro algoritmos de función de valor propuestos, con cuatro valores de ε distintos: un valor de alta exploración $\varepsilon = 0,25$, un valor de exploración media $\varepsilon = 0,5$, un valor de exploración baja de $\varepsilon = 0,75$ y un valor de muy baja exploración de $\varepsilon = 0,9$. En la Figura 24 se observa que el valor $\varepsilon = 0,75$ obtiene los mejores resultados en cuanto a promedio de latencia. Además, si analizamos la convergencia en la Figura 25, se puede ver que efectivamente que el valor de ε mencionado converge más rápidamente y a un mejor valor, en todos los algoritmos. Por otra parte, los valores más extremos $\varepsilon = 0,9$ y $\varepsilon = 0,25$, de muy alta y baja exploración, consiguieron los peores resultados. El valor medio $\varepsilon = 0,5$ tuvo buenos resultados, pero no los mejores. En conclusión, se decide utilizar un valor de exploración baja $\varepsilon = 0,75$.

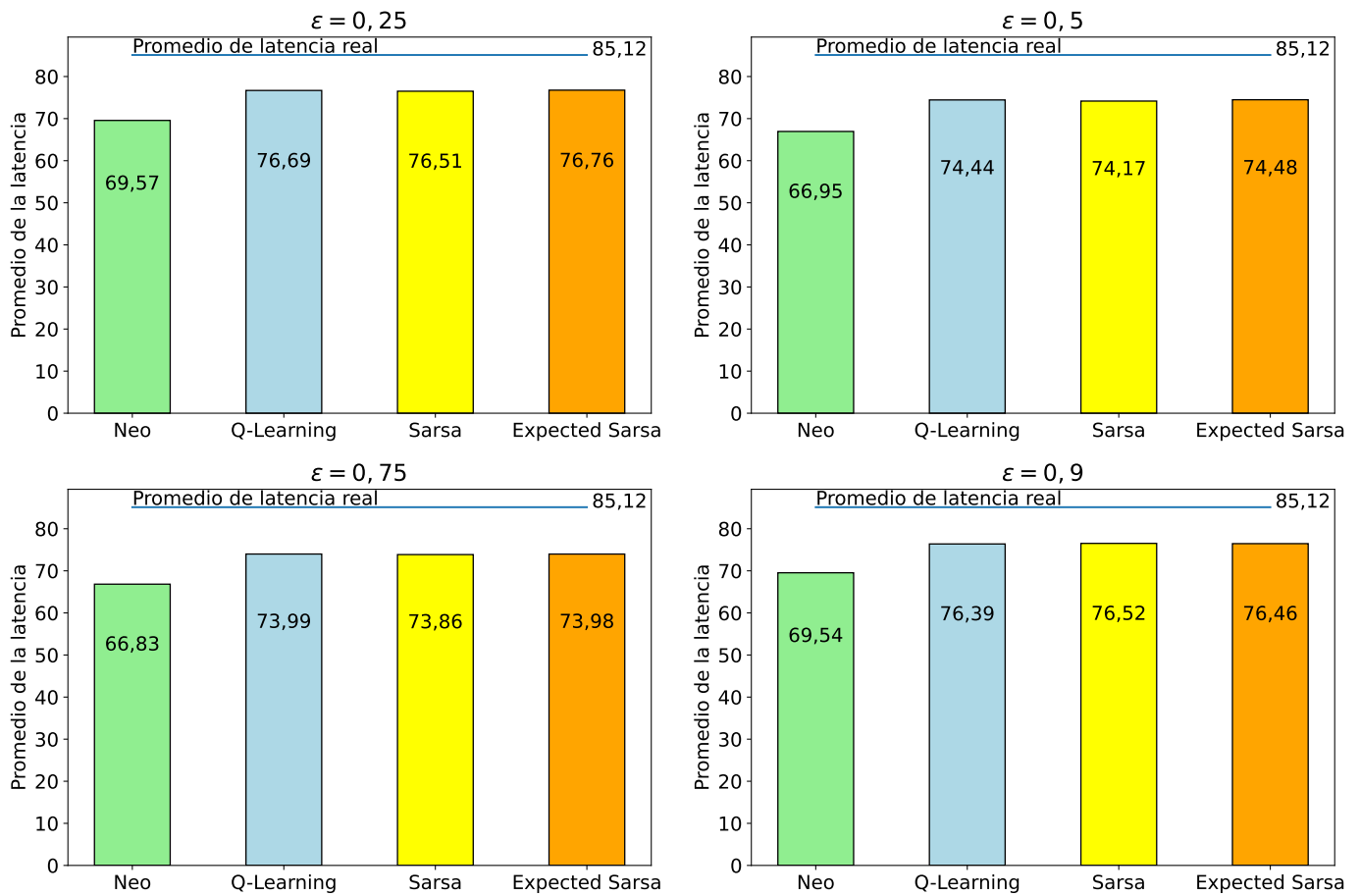


Figura 24: Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de ϵ .

Fuente: Elaboración propia.

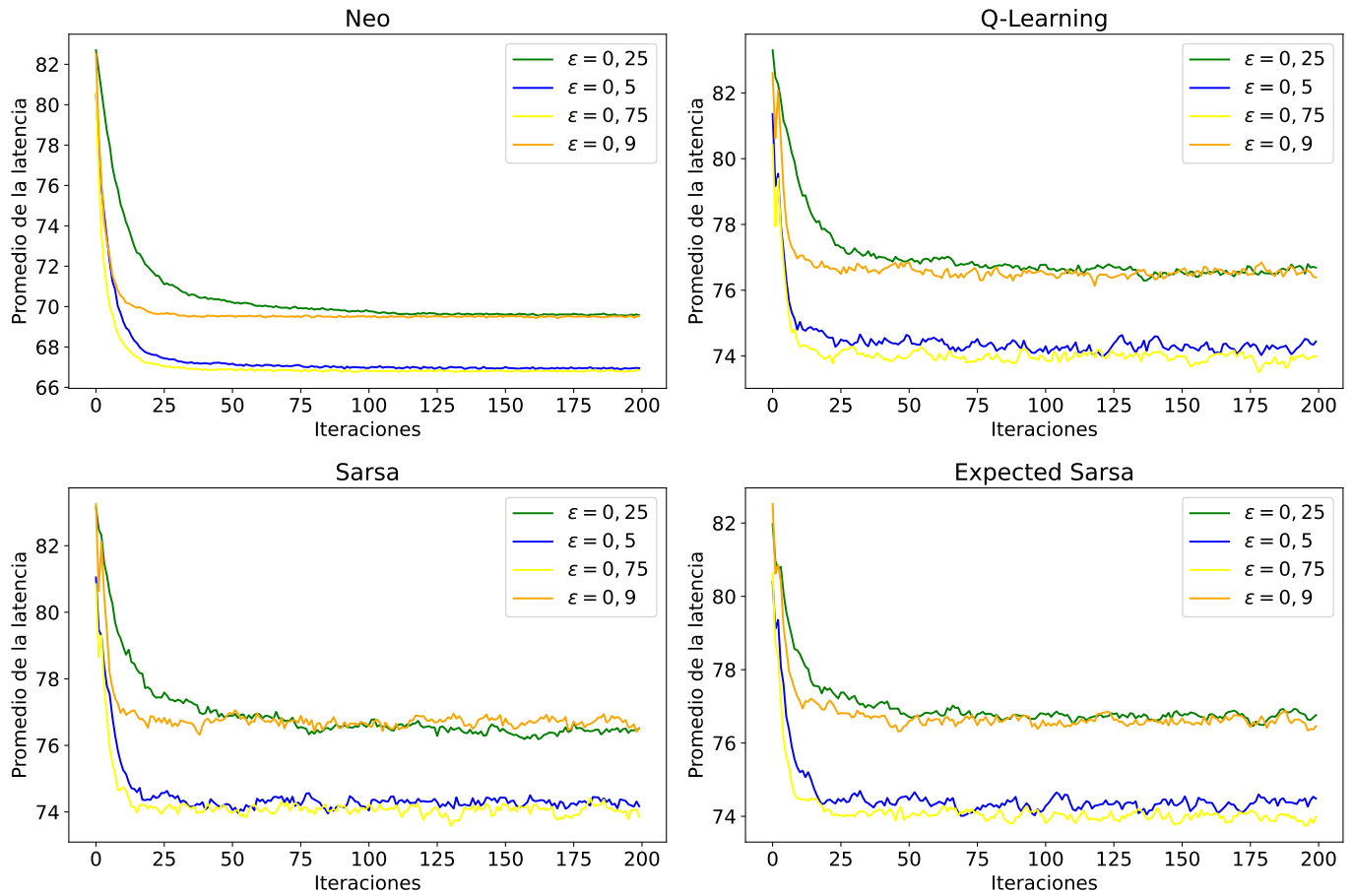


Figura 25: Convergencia de cada algoritmo con distintos valores de ϵ .

Fuente: Elaboración propia.

Tuning de tasa de aprendizaje α

La tasa de aprendizaje o tamaño del paso determina hasta qué punto la información recién adquirida anula la información antigua. Un factor 0 hace que el agente no aprenda nada (explotando exclusivamente el conocimiento previo), mientras que un factor 1 hace que el agente considere solo la información más reciente (ignorando el conocimiento previo para explorar posibilidades). En entornos totalmente deterministas, una tasa de aprendizaje de $\alpha_t \approx 1$ es óptima. Cuando el problema es estocástico, el algoritmo converge bajo algunas condiciones técnicas sobre la tasa de aprendizaje que requieren que disminuya a cero, donde en la práctica, a menudo se utiliza una tasa de aprendizaje constante y más baja, como $\alpha_t \approx 0,1$ para todo t [Sutton y Barto, 2018]. Similar al *tuneo* de hiperparámetros de ε , se entrenaron 12 modelos usando la base mencionada, 0,999 como valor del factor de descuento γ , y 0,75 como valor de exploración ε . Se utilizaron los tres algoritmos de TD, pero no se usó el algoritmo de Neo, pues este no depende de α para su aprendizaje. Los valores de tasa de aprendizaje α probados fueron: 0,9, 0,09, 0,009 y 0,999. En la Figura 26 se graficaron el promedio de latencia de cada modelo para cada tasa de aprendizaje comparándolos además con el promedio real de latencia de VOS, y en la Figura 27 se graficaron las convergencias de los modelos para cada tasa de aprendizaje. Los resultados indican que los α con valores menores al orden de magnitud 0,1 son los que más lento aprendían y los que a peores optimizaciones llegaron. En particular el valor de $\alpha = 0,009$ fue el peor de todos. Por su parte, no se apreciaron diferencias significativas entre un α alto de valor 0,9 y α de valor muy alto 0,999. Por tanto, se decide utilizar el valor $\alpha = 0,9$.

Tuning de factor de descuento γ

El factor de descuento γ determina la importancia de las recompensas futuras. Generalmente, se le asigna un valor entre 0 y 1. Un factor de 0 hará que el agente sea "miope" (o cortoplacista) al considerar solo las recompensas actuales, mientras que un factor que se acerque a 1 hará que se esfuerce por obtener una recompensa alta a largo plazo. Si el factor de descuento alcanza o supera 1, la política probablemente diverja. Para $\gamma = 1$, sin un estado terminal, o si el agente nunca llega a uno, las recompensas pueden volverse infinitas. Incluso con un factor de descuento solo ligeramente inferior a 1, el aprendizaje de la función Q conduce a la propagación de errores e inestabilidades [François-Lavet *et al.*, 2015] [Sutton y Barto, 2018]. Se entrenaron 18 modelos con seis valores de γ distintos. Además, únicamente se usaron los algoritmos TD para la función de valor, pues al igual que α , la Neo no depende de γ . Se usó la base mencionada, 0,9 como valor de tasa de aprendizaje α , y 0,75 como valor de exploración ε . Los valores de factor de descuento γ probados fueron: 0,0999, 0,25, 0,5, 0,75, 0,9 y 0,999. En la Figura 28 se observa el promedio de latencia de los algoritmos utilizando estos valores de *alpha*. En la Figura 29 se graficó la convergencia de estos valores de *gamma* en cada algoritmo. En ambos gráficos se observa que el valor $\gamma = 0,25$ obtuvo los peores resultados. Para el resto de valores de γ no existen diferencias significativas, pero 0,999 es levemente superior al resto. Por lo tanto, se decide usar 0,999 como valor de γ .

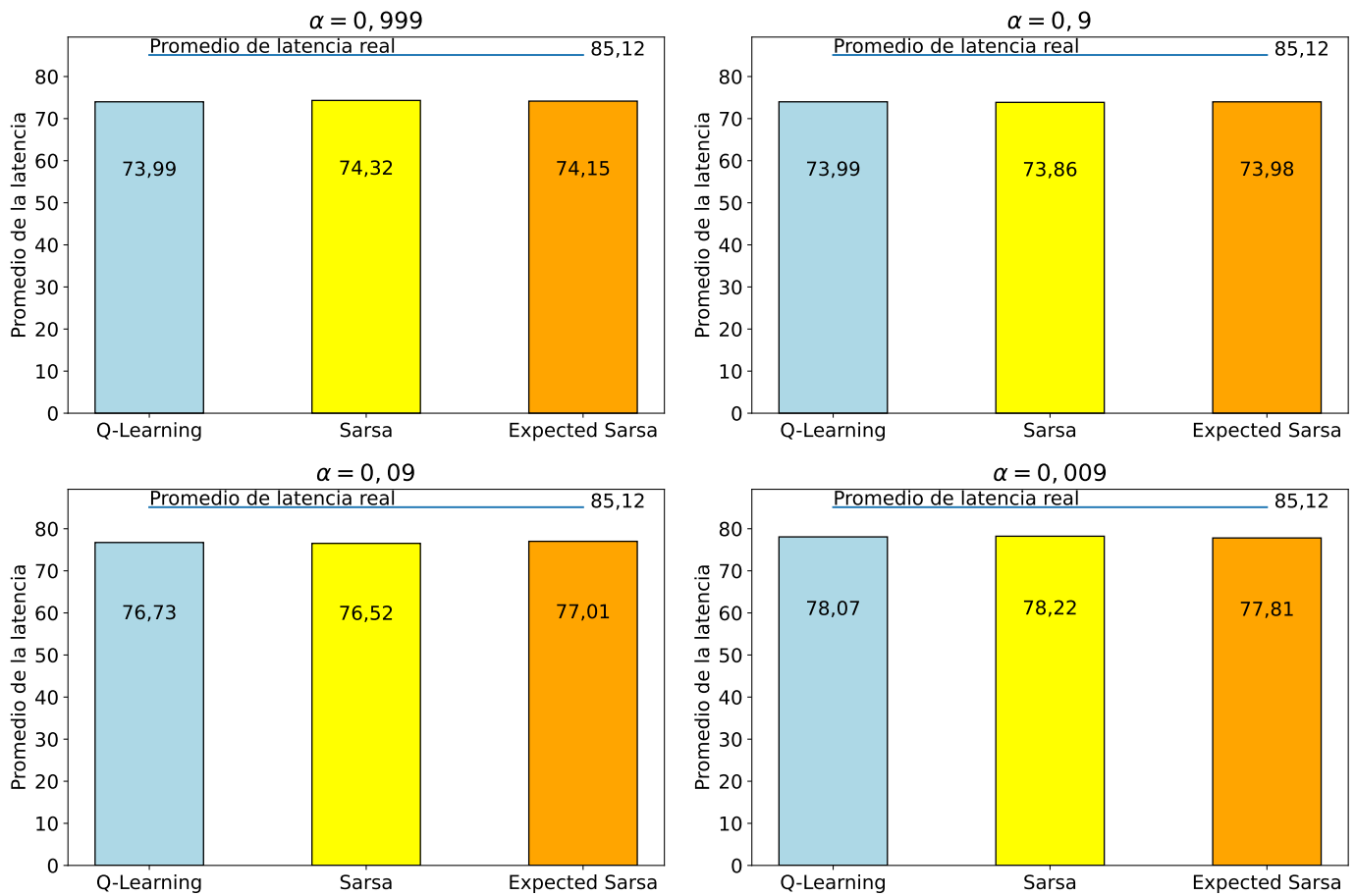


Figura 26: Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de α .

Fuente: Elaboración propia.

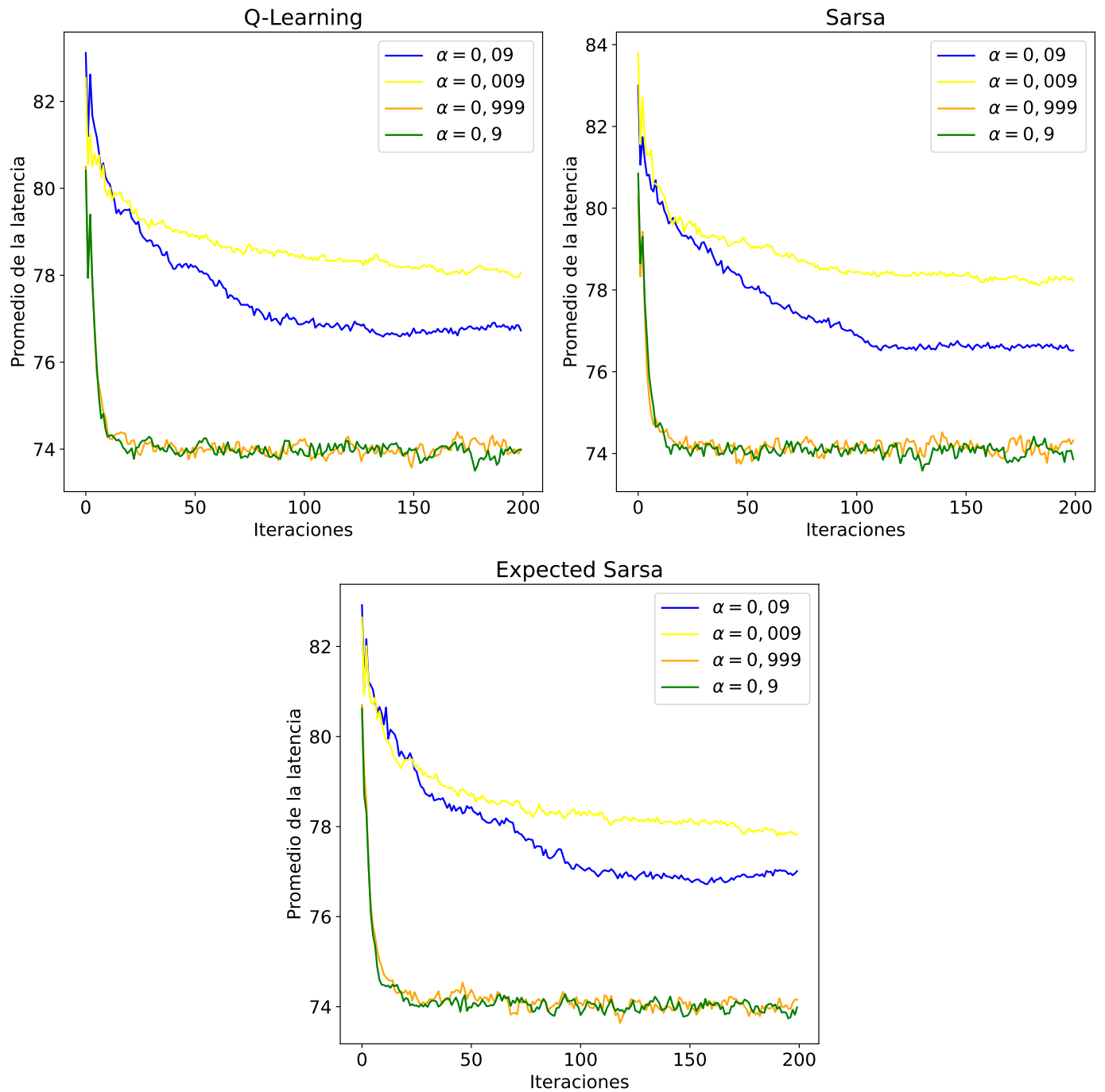


Figura 27: Convergencia de cada algoritmo con distintos valores de α .
Fuente: Elaboración propia.

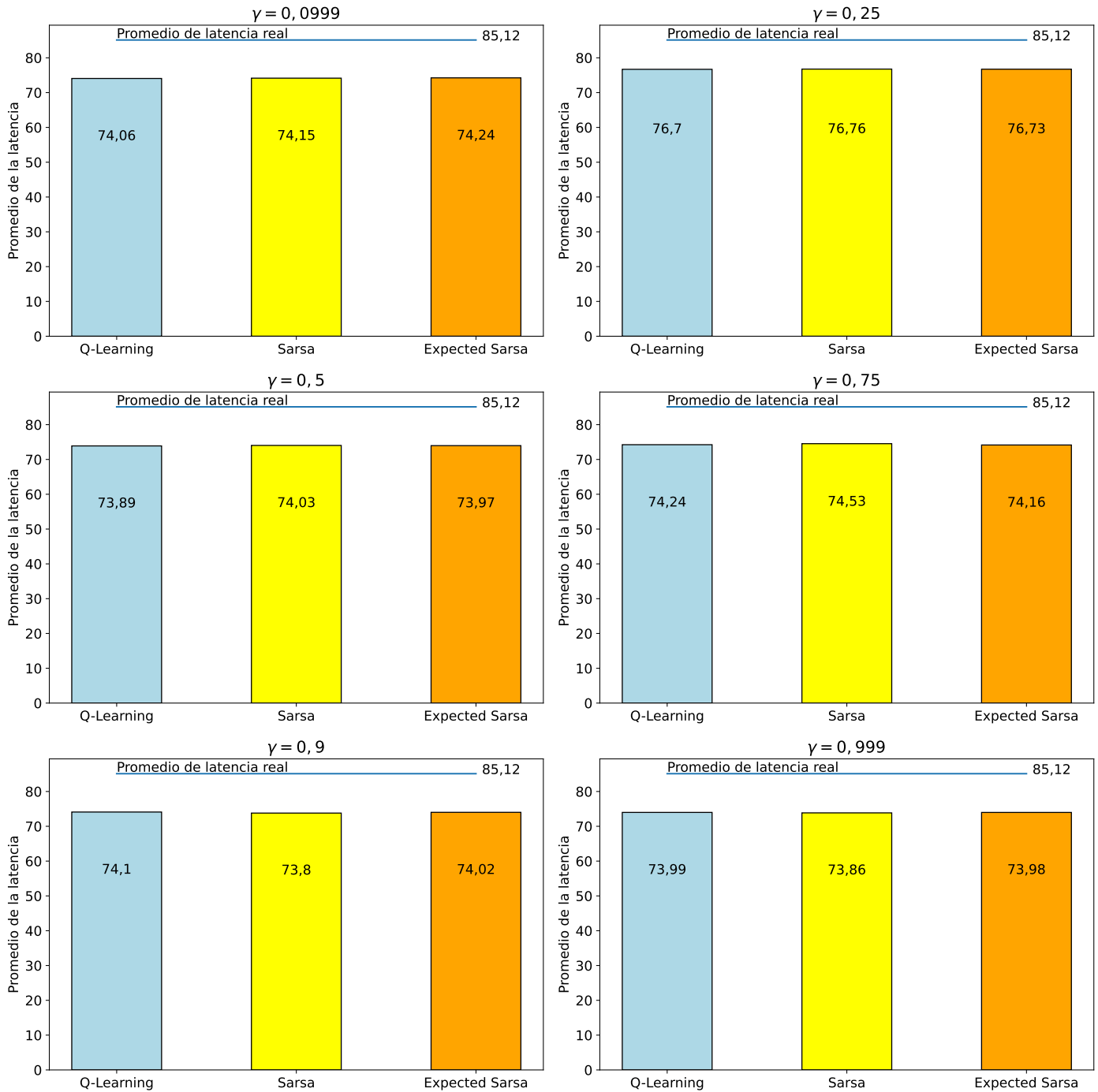


Figura 28: Promedio de latencia de cada algoritmo comparado con el promedio de latencia real, para distintos valores de γ .

Fuente: Elaboración propia.

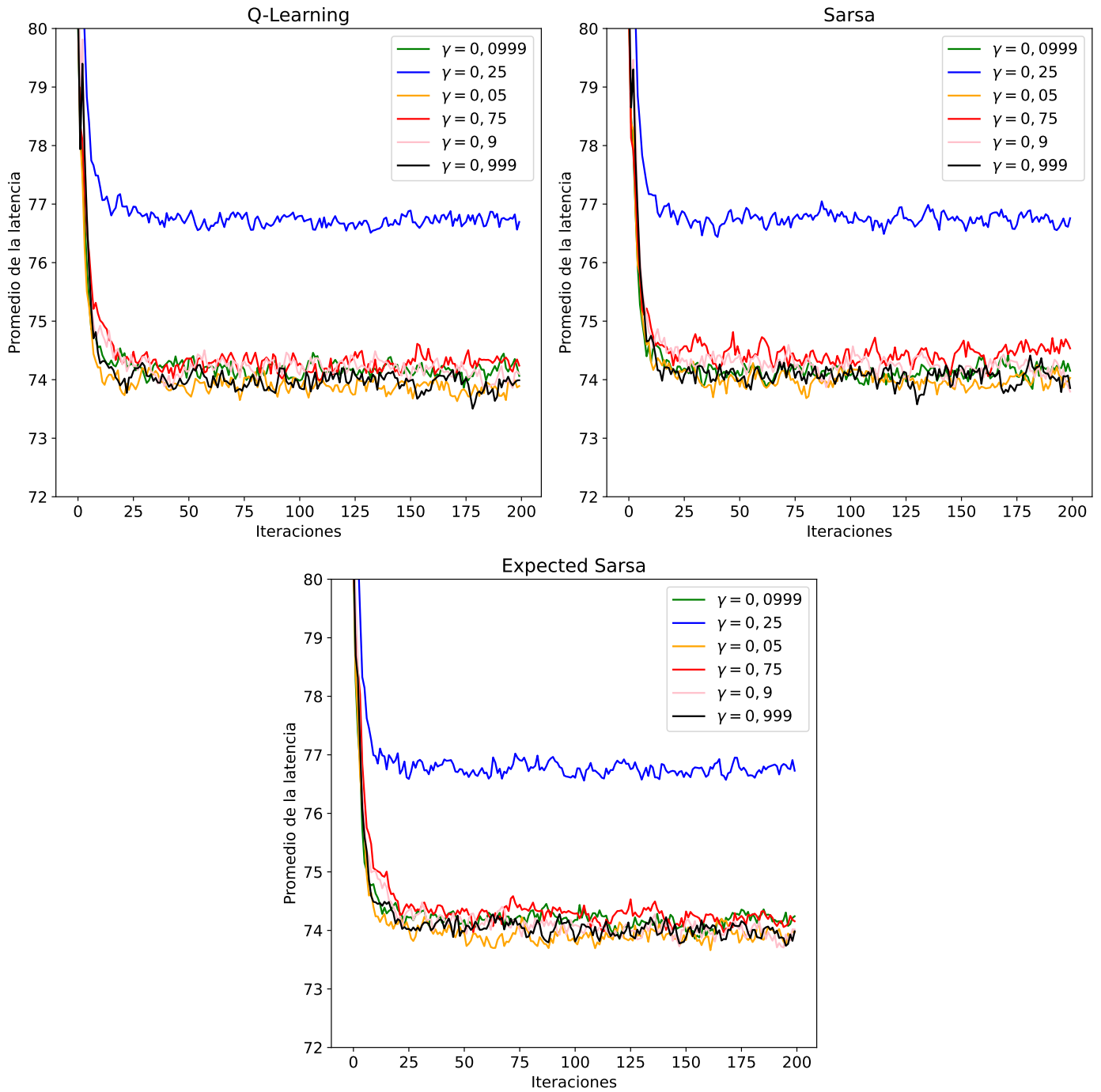


Figura 29: Convergencia de cada algoritmo con distintos valores de γ .
Fuente: Elaboración propia.

4.2.2. Análisis a los mejores modelos DRL obtenidos

Se escogieron los cuatro modelos con mejor comportamiento en el promedio de latencia y en la convergencia tras el tuneo de hiperparámetros, usando cada uno de los cuatro algoritmos de función de valor propuesto. Para el análisis de estos modelos de DRL, se realizaron distintos experimentos comparativos, tanto entre ellos como también con los valores reales de VOS. Se ejecutaron 200 iteraciones en cada consulta, para cada modelo. Los hiperparámetros escogidos tras el *tuneo* anteriormente detallado fueron $\varepsilon = 0,75$ para el algoritmo ε -greedy, un valor de tasa de aprendizaje $\alpha = 0,9$ y un factor de descuento $\gamma = 0,999$.

En la Figura 30 se graficó el promedio de los cuatro modelos, tanto en general como por iteración para analizar la convergencia. En cuanto al promedio general de latencia se observa que el algoritmo Neo es claramente superior a los TD. Esto queda aún más en evidencia si observamos el gráfico de convergencias, pues se observa claramente como Neo, no solo diverge más rápido que los otros algoritmos, sino que converge a un valor mucho mejor y con más estabilidad (los otros algoritmos tienen muchas crestas en su curva, mientras que la de Neo es lisa). Descartando Neo, se aprecia que los otros algoritmos no muestran diferencias significativas en cuanto a promedio de latencia. De todas maneras, en todos los casos los modelos promediaron una latencia mejor que la original de VOS.

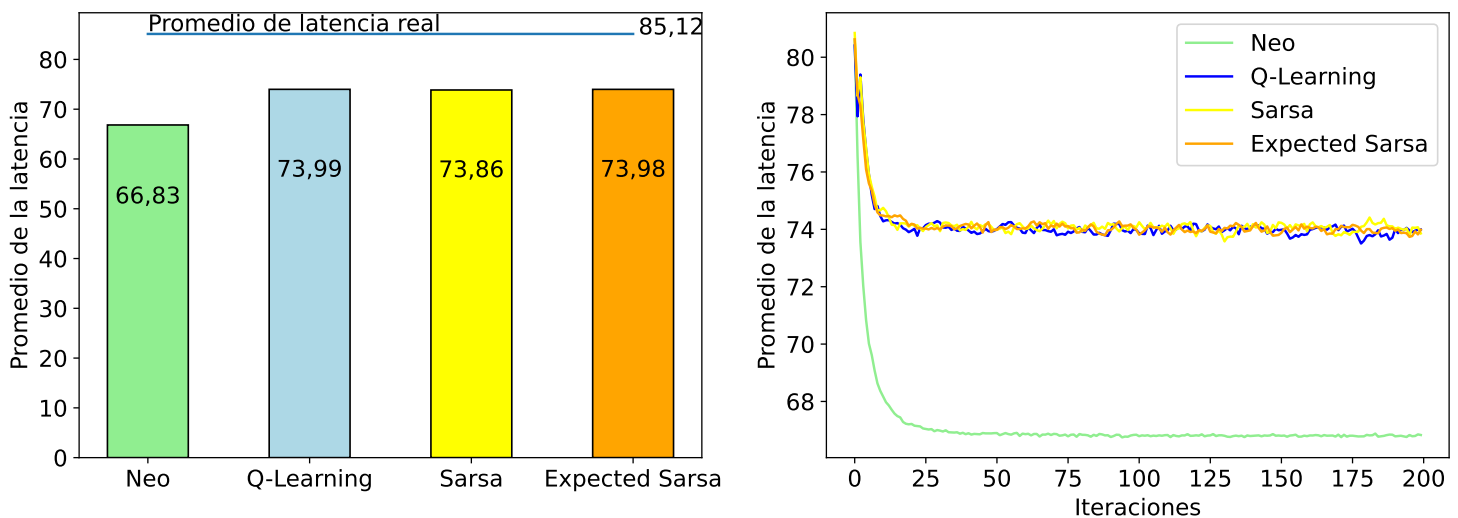


Figura 30: Promedio de latencia de los mejores modelos DLR, junto con la convergencia de cada uno de estos.

Fuente: Elaboración propia.

La Figura 31 muestra el promedio de latencia en cada algoritmo separados según el número de JOINS de las consultas. Podemos concluir que independiente del número de JOINS el modelo con algoritmo Neo tiene el menor promedio de latencia de las consultas, y que todos los modelos superan a VOS en este aspecto. Por otra parte, se observa que entre los algo-

ritmos TD, Sarsa es ligeramente superior en consultas menos complejas con menor número de JOINS, pero ocurre lo contrario cuando estos van aumentando.

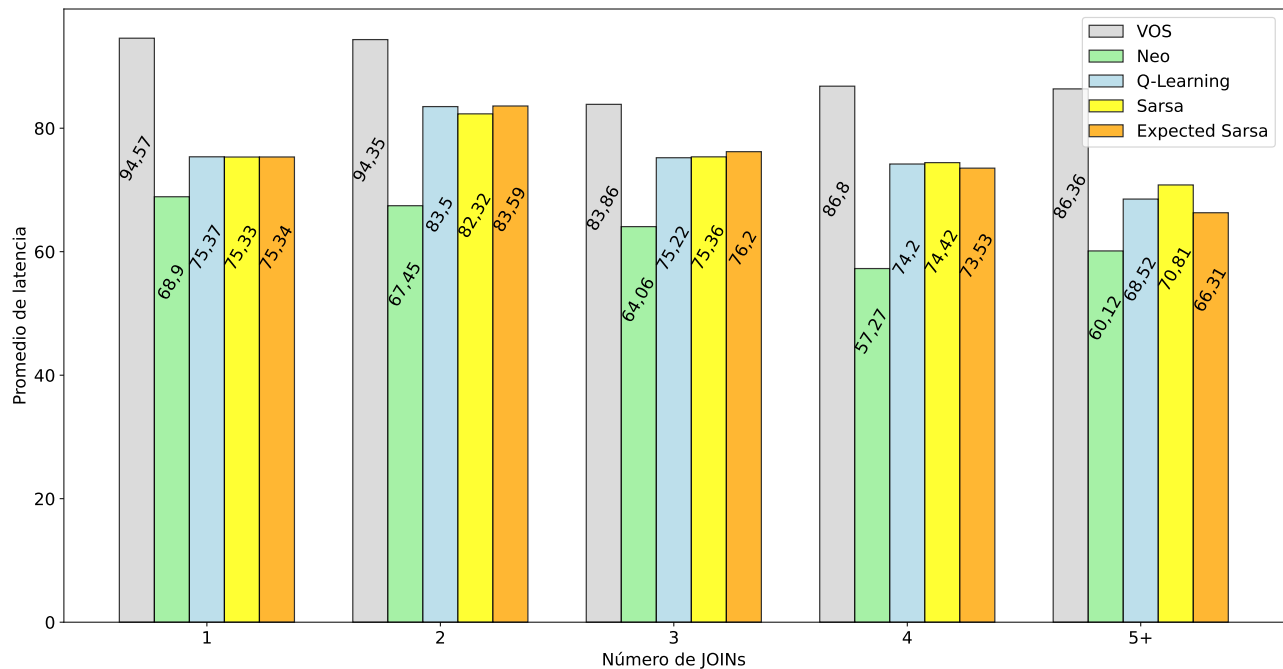


Figura 31: Promedio de latencia de los mejores modelos DLR, separados por numero de JOINS.

Fuente: Elaboración propia.

A lo que respecta al análisis de errores, se observa en la Figura 32 que el modelo Neo logró optimizar la mayor cantidad de consultas posibles, es decir, consiguió encontrar mejores árboles binarios, es decir, mejores planes de ejecución de consultas que los otros algoritmos. Si expandimos este análisis a los errores de los modelos por rango de consulta, se observa en la Figura 33 que todos los algoritmos de optimización funcionan mejor con consulta de latencia alta. Esto es una buena noticia, ya que es justo lo que se buscaba (optimizar consultas interesantes). Mejor aún, los errores son cada vez menores a medida que la latencia de la consulta aumenta, donde en el peor de los casos los modelos encuentran un plan de consultas igual (o extremadamente similar) al propuesto por el optimizador. Se aprecia además que Neo funciona mejor en todos los rangos de latencia, mientras que entre los otros algoritmos no se aprecian diferencias significativas. Finalmente, en la Figura 34 se graficaron la cantidad de errores por número de JOINS en cada modelo. Como es la tónica, Neo, independiente del número de JOINS es mucho mejor que cualquier otro algoritmo. Por otra parte, los algoritmos TD registran muchos errores en consultas de dos y tres JOINS al presentar una cantidad importante de consultas que fueron empeoradas en latencia. Lamentablemente, no se pueden sacar más conclusiones de este gráfico pues el número de consultas era bastante desbalanceado respecto al número de JOINS.

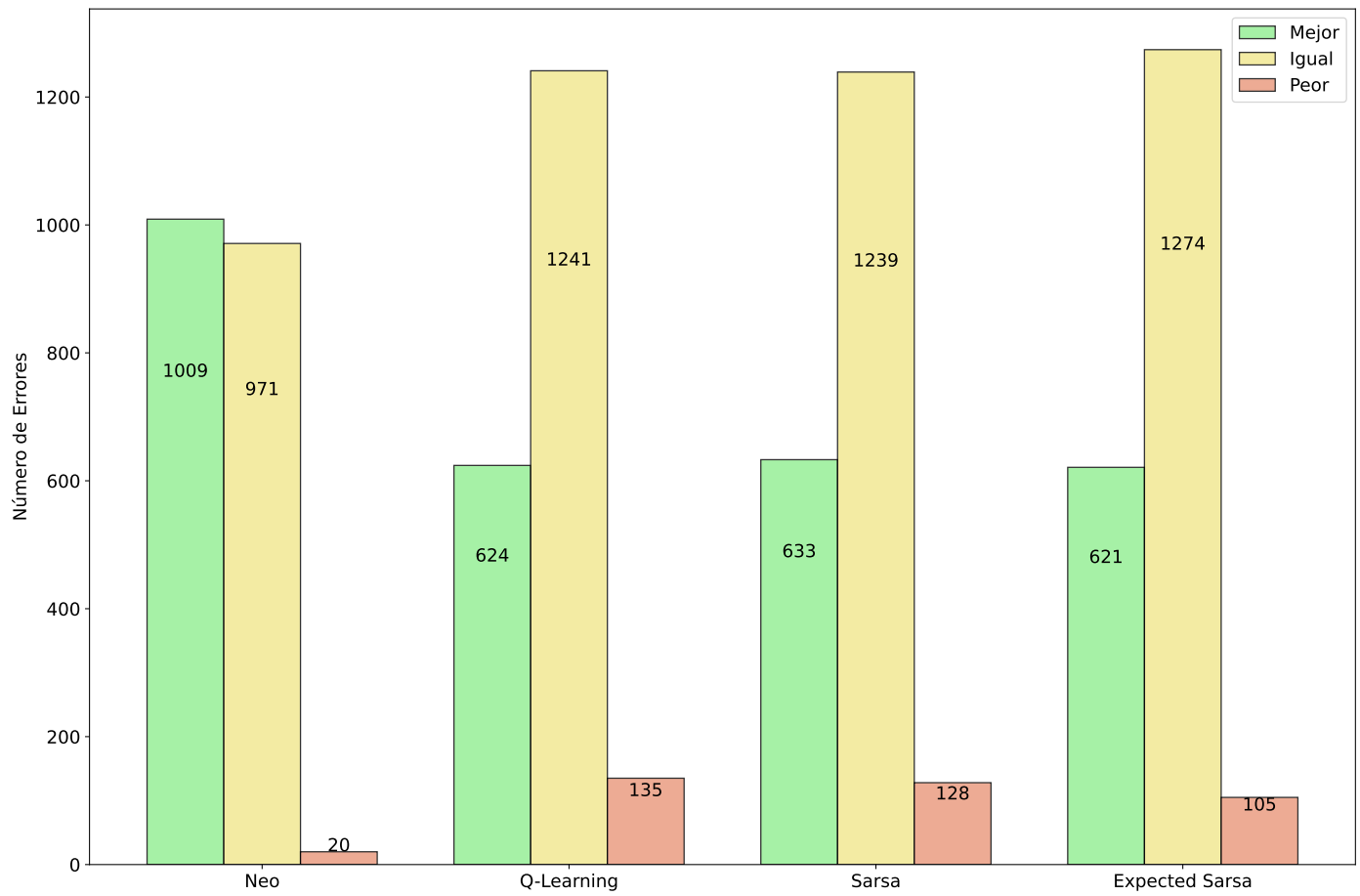


Figura 32: Errores totales de cada modelo DLR.
Fuente: Elaboración propia.

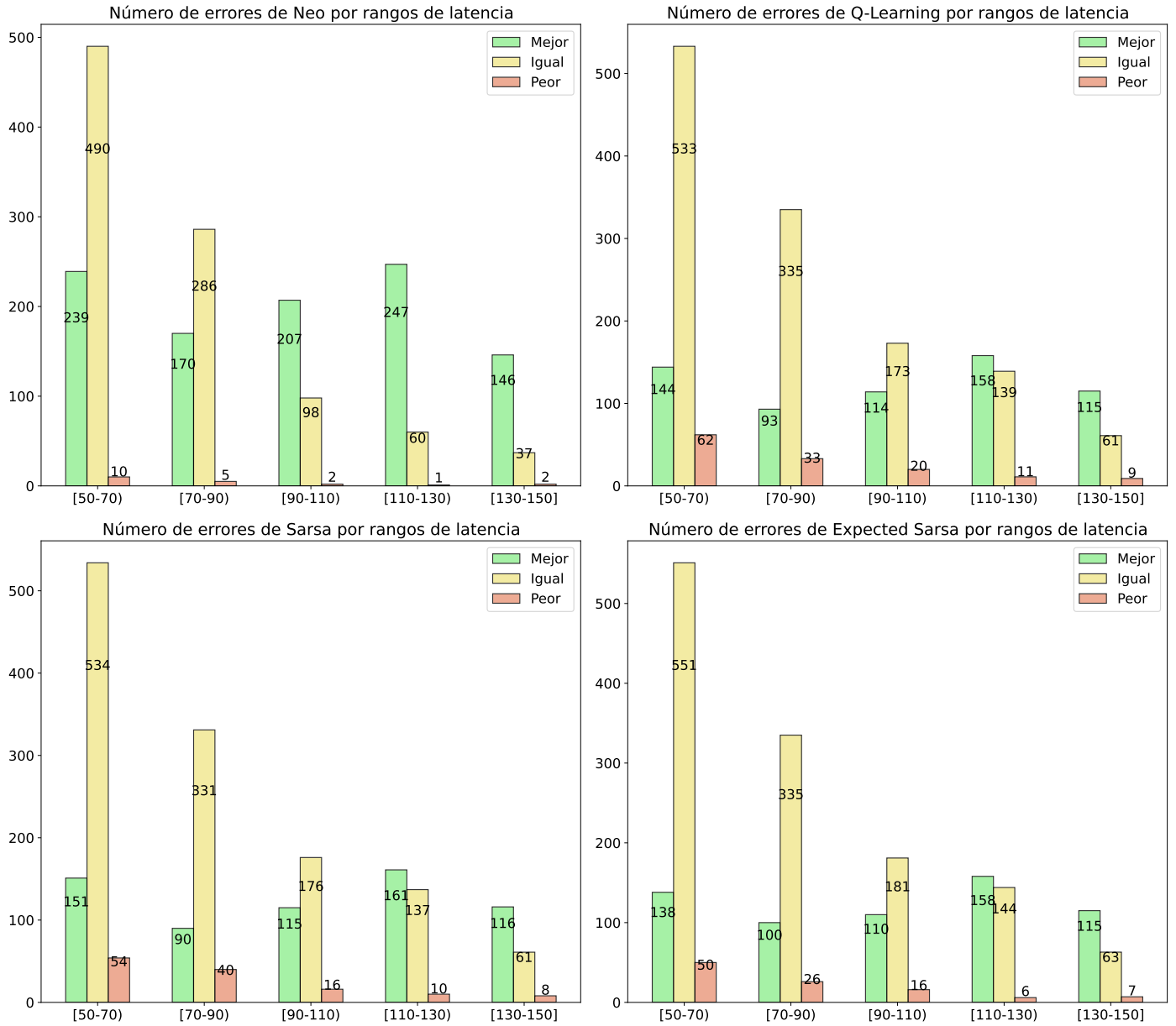


Figura 33: Errores de cada modelo DLR separados en distintos rangos de latencia.
Fuente: Elaboración propia.

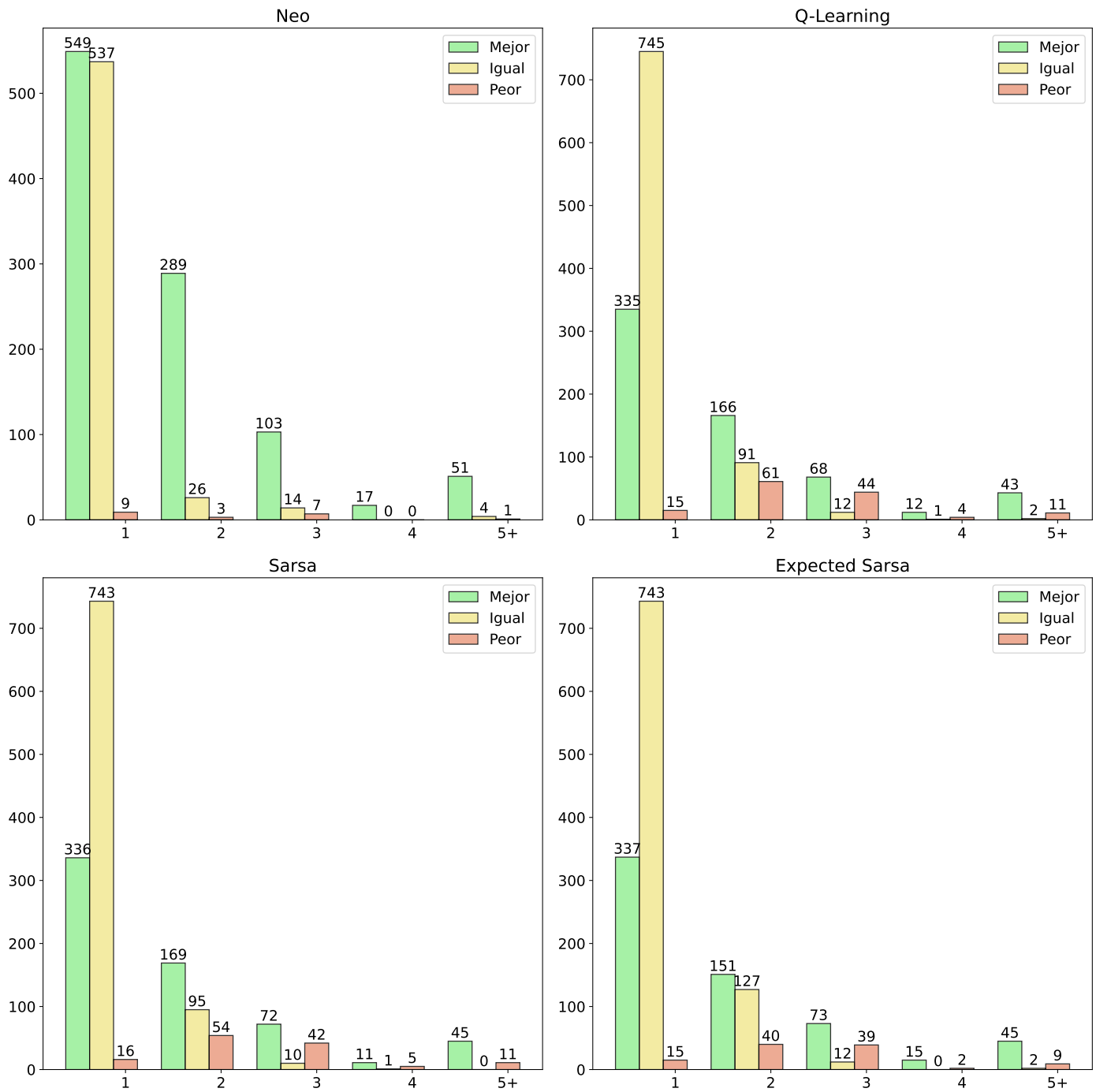


Figura 34: Errores de cada modelo DLR separados por número de JOINS.

Fuente: Elaboración propia.

CAPÍTULO 5

CONCLUSIONES

Con la propuesta realizada se ha cumplido el objetivo principal así como los objetivos específicos. Principalmente, se logró optimizar consultas de SPARQL con distintos modelos de *Deep Reinforcement Learning*, una técnica muy interesante que junta elementos de *Deep Learning* y de *Reinforcement Learning*. Los resultados obtenidos muestran que estos modelos lograron generar planes de ejecución mejores que el los planes de ejecución de consultas de *Virtuoso Open Source*. En particular, el modelo que utiliza el algoritmo de Neo propuesto precisamente por ellos en [Marcus *et al.*, 2019], resulto ser altamente superior a los algoritmos de RL clásicos del tipo TD: *Q-Learning*, *Sarsa* y *Expected Sarsa*. No obstante, al ser un problema no tan abordado en la literatura, existen varios problemáticos detalles que contaminan los resultados finales, pero que abren varias **oportunidades de mejora** al trabajo realizado. A partir de esta memoria, existen varios **trabajos futuros** que podrían generarse utilizando como base las investigaciones, modelos y/o resultados obtenidos en esta memoria.

5.1. Conclusiones Generales

- Uno de los objetivos específicos era la de explorar nuevas formas de representar los datos de consultas SPARQL. Es por eso que se decidió utilizar un gestor de base de datos no muy utilizado hasta ahora para investigaciones de este tipo, como lo es VOS. No obstante existió el problema de que VOS no tiene una manera de extraer información con respecto al plan de consultas más que la sentencia PROFILE. Es por eso que se desarrolló un extractor de características para VOS, el cual es capaz de construir el árbol de ejecución para una consulta determinada, así como los subárboles que representan las subconsultas. Este lee las sentencias PROFILE que entrega VOS y encuentra distintos patrones para detectar las consultas, los predicados, etc.
- Para poder desarrollar unos modelos similares de DRL al propuesto por Neo para consultas SQL [Marcus *et al.*, 2019], es necesario desarrollar y entrenar un modelo supervisado de DL que permita predecir la latencia de las consultas, en este caso, de SPARQL. Como la predicción de latencia de consulta no es un problema generalizado por la comunidad de DL, no existe un estándar respecto a que tipo de redes neuronales utilizar, a diferencia de otros campos. El uso de *TreeConv*, propuestos en Neo resulto muy bien. Se decidió utilizar y adaptar el modelo propuesto por [Casals *et al.*, 2020] el cual estaba ya adaptado a SPARQL, con muy buenos resultados. No obstante, al ser un *dataset* y un gestor de base de datos distintos al trabajo mencionado, se realizaron algunos experimentos *tuneando* algunos hiperparámetros. Los resultados fueron satisfactorios, aunque con algunos detalles que se verán en la subsección de más adelante.
- Respecto a la optimización de consultas SPARQL, fue necesario tener en cuenta:

- Replantear el problema como un MDP, en donde el árbol se va rearmando, donde cada subconsulta escogida era una acción y cada subárbol era un estado. Las recompensas por su parte se obtenían utilizando el modelo de predicción de latencia.
- Una propiedad de SPARQL muy importante, el cual es el hecho de que dos subconsultas de BGPs distintos no pueden entremezclarse hasta que se procesen todas las subconsultas de alguno de los dos primeros. Esto afecta principalmente en la elección de una acción, pero también afectaba algunos algoritmos que necesitaban calcular el máximo valor de una acción en un estado particular (utilizando la política).

Teniendo estos dos conceptos en cuenta, se logró diseñar una base para modelos de DRL, similar a la que utiliza Neo, en donde cada consulta puede encontrar un plan de consultas optimizado (construyendo un nuevo árbol de ejecución).

- Se logró diseñar, implementar y comparar cuatro modelos de RL utilizando la base mencionada anteriormente, y cuatro algoritmos de función de valor, las cuales son las que se implementó en Neo y tres de tipo *Temporal Difference* o TD: *Q-Learning*, *Sarsa* y *Expected Sarsa*.

5.2. Conclusiones de resultados

5.2.1. Resultado del modelo de predicción de latencia desarrollado

- Se hizo un *tuning* de hiperparámetros en el modelo de predicción de latencia. En particular se *tunearon* la función de optimización, la tasa de aprendizaje y la complejidad del modelo cambiando las capas ocultas. Las métricas utilizadas fueron RMSE y MAE. Los hiperparámetros *tuneados* fueron:
 - Se probaron los optimizadores *Adam*, *AdaGrad* y *SGD*. Este último mencionado, fue por lejos el que peor funcionó, convergiendo lentamente y a valores de RMSE y MAE muy altos en comparación a los otros 2. Los otros dos optimizadores que se probaron mostraron buenos resultados, pero *AdaGrad* fue más inestable que *Adam*, por lo que se escogió este último.
 - Se probaron tasas de aprendizaje con los valores 0,0015, 0,00015, 0,000015 y 0,0000015. Los valores extremos 0,0015 y 0,0000015 fueron las tasas con peores resultados. Las tasas con valores 0,00015 y 0,000015 obtuvieron buenos resultados, siendo el primero de estos, el mejor, y por ende, el elegido.
 - Se probaron seis distintas arquitecturas de redes neuronales, haciendo varias combinaciones de tamaños y cantidades de capas ocultas, tanto para las capas de *Fully Connected Layers*, como para las capas de *TreeConv*. No se apreciaron mayores diferencias.

Los hiperparámetros escogidos tras el *tuning* fueron: *Adam* como función de optimización, 0,00015 como tasa de aprendizaje y 180 épocas de iteración. Con respecto a los hiperparámetros asociados a la arquitectura de la red neuronal, se usaron tres capas de *Fully Connected Layer* con tamaños (32,64,128) y función de activación *ReLU*, y tres capas de *TreeConv* con tamaños (512,256,128) y función de activación *LeakyReLU*. La arquitectura final de la red neuronal construida es muy similar a la Figura 10 presentada en el Capítulo 1, con la diferencia de que los datos de entrada tienen un tamaño de 5, en vez de 64, ya que las características a nivel de consulta son menos que las utilizadas en [Marcus *et al.*, 2019] y [Casals *et al.*, 2020].

- Se analizó el mejor modelo de predicción de latencia obtenido. Se comparó la latencia real de las consultas versus la latencia obtenida por el modelo, en todos los conjuntos de datos (entrenamiento, validación y prueba). Los resultados fueron que más del 70 % de las consultas tuvieron una predicción buena o aceptable. No obstante, también los resultados muestran que el modelo falla más en la predicción de consultas de baja latencia.
- Los valores de RMSE y MAE del mejor modelo obtenido fue de 21 y 15 para el conjunto de entrenamiento, y 27 y 20 para el conjunto de validación, respectivamente. Si bien no son malos resultados, pueden ser mejores, y se concluye la existencia de un pequeño grado de *overfitting*.

5.2.2. Resultados de los modelos de optimización de consulta desarrollados

- Al igual que en el modelo de predicción de latencia, se hizo un *tuning* de hiperparámetros. En este caso se probaron el valor ε para el algoritmo de selección de ε -greedy, la tasa de aprendizaje de RL α y el factor de descuento γ .
 - Los valores de ε escogidos fueron 0,25, 0,5, 0,75 y 0,9, que aseguraron probar distintos niveles de exploración. Se probaron estos valores en 16 modelos, pues cada valor de ε se probó en los cuatro algoritmos DR escogidos en esta memoria. Los resultados mostraron que los valores extremos de muy alta y muy baja exploración 0,25 y 0,9 fueron los de peores, tanto en latencia promedio optimizada como en convergencia. Mientras que los valores intermedios 0,5 y 0,75 mostraron mejores resultados. El mejor entre estos dos fue el valor de exploración baja, $\varepsilon = 0,75$.
 - Para la tasa de aprendizaje α se probaron tres valores de distintos órdenes de magnitud y uno muy cercano a uno, los cuales son: 0,999, 0,9, 0,09 y 0,009. En esta oportunidad no se entrenó el modelo Neo pues este no es afectado por α . En total se evaluaron 12 modelos. Los valores de α muy lejanos a uno fueron los que dieron peores resultados. Mientras que los otros dos valores obtuvieron mucho mejor rendimiento. Particularmente, el valor $\alpha = 0,9$ fue levemente mejor que 0,999.

- Con respecto al factor de descuento γ se probaron seis valores en los modelos que usaban algoritmos TD, pues al igual que con la tasa de aprendizaje α Neo no es afectado por γ . Así que en total se probaron 18 modelos. Los valores escogidos de γ fueron: 0,0999, 0,25, 0,5, 0,75, 0,9 y 0,999. Salvo por $\gamma = 0,25$, que fue el peor de todos con diferencia, no se apreciaron diferencias significativas con los otros valores.

Tras el *tuneo* de hiperparámetros, los valores escogidos fueron: $\varepsilon = 0,75$, $\alpha = 0,9$ y $\gamma = 0,999$.

- Se evaluaron los mejores modelos con distinto algoritmo como función de valor. Los resultados mostraron una clara superioridad del algoritmo Neo tanto en promedio de latencia como en la convergencia. Evaluando el promedio de latencia por número de JOINS, Neo también resultó ser el mejor de todos con cualquier cantidad de estos (siempre limitado al *dataset*, obviamente).
- Todos los algoritmos obtuvieron planes de consulta que tienen mejor promedio de latencia que los planes de consulta de VOS.
- Con respecto al número de errores por algoritmo, Neo también fue superior en ese aspecto, pues fue el que menos errores tuvo, pero además fue el que tuvo más acierto, ya que logro optimizar la mayoría de las consultas del *dataset*. Por el contrario, los otros algoritmos lograron la demasiadas veces encontrar el mismo plan de consultas que el que ejecutaba VOS. De todas maneras, lograron optimizar varias consultas.
- Analizando el número de errores por rango de latencia, nos encontramos con buenas noticias, pues la mayoría de las consultas optimizadas se encontraban en los rangos de latencia relativamente alta, mientras que para las consultas de latencia más baja, como es de esperar la mayoría encontró el mismo *dataset*. La excepción a esto fue el algoritmo Neo, el cual fue al igual que los demás, más efectivo en consultas de alta latencia, pero incluso en las de baja latencia mostró buenos resultados.
- Con respecto al número de errores por número de JOINS, Neo nuevamente fue superior a todos. Por otro lado, los otros algoritmos presentaron una cantidad de errores significativos cuando la cantidad de JOINS eran de uno o dos, aunque de todas maneras siguen siendo minoría estos errores en comparación a los aciertos e igualdades. Lamentablemente no se pudieron sacar más conclusiones pues la mayoría de consultas tenían uno o dos JOINS en su plan de consultas, lo que significa un *dataset* desbalanceado en ese aspecto.
- En conclusión, se lograron realizar cuatro modelos DRL con cuatro algoritmos de función de valor distintos, donde todos obtuvieron en la mayoría de las consultas del *dataset*, planes de consultas con mejor latencia que los que propone VOS. Neo en particular resulto ser el mejor de todos.

5.3. Oportunidades de mejora

- Un problema importante cuando se empezó a entrenar el modelo de predicción de latencia fue el *dataset*, pues habían demasiadas consultas de muy baja latencia, incluida muchas con latencia 0. En resumen, el mayor problema que tenía es que era demasiado disperso y desbalanceado. Además, por la naturaleza de la memoria, se agregan al *dataset* base muchas subconsultas, donde muchas de ellas también tenían muy bajas latencias, provocando un desbalance mucho mayor. Se hicieron esfuerzos para enfrentar este problema reduciendo la latencia posible tanto de consultas enteras como de subconsultas, logrando minimizar el problema. No obstante, existe una gran oportunidad de mejora pues se evaluaron consultas de máximo 200 ms, un valor muy tolerable de esperar en la vida real. Por tanto, si se pudiera mejorar el *dataset* de consultas, logrando que sea más balanceado y agregando consultas más interesantes de optimizar. Algunos problemas que genera a la larga esto son:
 - Dificultad a la hora de entrenar el modelo (principalmente un pequeño sobreajuste).
 - Existen consultas que no tienen sentido optimizar pues los modelos de DRL llegan al mismo plan de consulta (lo cual ocurre porque hay muchas consultas demasiado simples).
 - Se generan consultas que los optimizadores de DRL empeoran debido a que la predicción de latencia de consultas y subconsultas cuando se obtiene la recompensa no es un valor correcto. A su vez también pueden existir “árboles sobrevalorados”, es decir, que el optimizador los predice mejor de lo que realmente son. No son muchas las que están en esta situación pero existen.
 - El análisis de los modelos DRL por número de JOINS se vio mermado pues la mayoría de estos tenían 1 o 2, lo cual es una lástima pues hubiera sido interesante ver el comportamiento de los modelos DLR con consultas más complejas en ese aspecto.
- Si bien puede deberse al problema del *dataset*, lo cierto es que el modelo de predicción de latencia tiene un pequeño sobreajuste, también llamado *overfitting*. Esto es en pocas palabras, que el modelo es muy bueno para la predicción de latencias en el conjunto de entrenamiento, pero cae en rendimiento respecto a otros conjuntos (como el de test, o el de validación). Sería interesante hacer modificaciones más profundas al modelo y ver si es posible corregir este sobreajuste.
- Agregando a lo anterior, también modificando el modelo de predicción de latencias podría mejorarse los valores de MSE, RMSE y MAE, que si bien no son malos, aún pueden mejorarse bastante.
- No es muy importante, pero a nivel de código los modelos de RL se realizaron con programación funcional¹¹. A medida que se avanzó durante la memoria el autor se dió

¹¹Código fuente en este *notebook*, en la sección de *Reinforcement Learning*.

cuenta de que hubiera sido mucho mejor modelarlos utilizando programación orientada a objetos, es decir, usando clases. Por tanto, una buena mejora sería adaptar el código de muchas funciones a una sola clase.

5.4. Trabajos futuros

Teniendo en cuenta las conclusiones generales, las conclusiones de resultados y las oportunidades de mejora, este trabajo deja la puerta abierta para que otras investigaciones similares puedan surgir:

- Identificar características en consultas SPARQL ha sido bastante trabajado por ejemplo en [Casals *et al.*, 2020] o en [Hasan y Gandon, 2014]. No obstante muchas de estas no eran aplicables a este trabajo pues en este caso se necesitaban características de subconsultas (y que por ende se extrapolen a características de consultas). Por tanto, una buena idea sería identificar nuevas características pero en las subconsultas, ya que en este trabajo se utilizaron características muy básicas y simples. Aumentar la cantidad de características podría ayudar a mejorar el modelo de predicción de latencia, y, por tanto, a lograr optimizaciones más precisas.
- Con respecto a la predicción de latencia de consultas, los modelos con redes neuronales del tipo *Tree-Structured LSTM* han demostrado su efectividad en el procesamiento semántico del lenguaje natural[Tai *et al.*, 2015]. Sería interesante explorarlas e implementarlas en la predicción de latencia SPARQL.
- Proponer como podría implementarse estos modelos DRL en un ambiente real. Es decir, como un gestor de base de datos podría utilizar los modelos propuestos para mejorar efectivamente la latencia de las consultas.
- Una desventaja importante es que los modelos diseñados optimizan las consultas “uno a uno”. Esto quiere decir que cada consulta tiene su propia política, el cual tendrá un tamaño distinto dependiendo del plan de ejecución. La implementación de un modelo que pueda optimizar cualquier consulta independiente del tamaño, es decir, una política “universal”, sería muy interesante. Una idea sería usar una lógica similar a las *TreeConv*, ampliando con nodos nulos los árboles de planes de ejecución de consultas del *dataset* hasta que alcancen el tamaño de uno de mayor tamaño.

ANEXOS

5.1. Pseudocódigos para distintos algoritmos de RL

En cuestión estos pseudocódigos son iguales a la Figura 16, pero especificando como obtienen la recompensa y como calculan la función de valor.

Para la **función de valor de Neo**:

```

Hiperparámetros de entrada:  $\varepsilon \in (0, 1)$ , num_iteraciones  $\leftarrow 200$ 
num_acciones = length( $A$ )
 $Q[s, a] \leftarrow \text{zero\_matrix}([\text{num\_acciones}, \text{num\_acciones}]), \forall s \in S, a \in A$ 
 $A_{\text{disponibles}} \leftarrow A, \forall a \in A$ 
for iteracion in num_iteraciones:
    Estado_actual  $\leftarrow \text{empty\_list}$ 
    Acción_escogida  $\leftarrow \varepsilon\text{-greedy}(\text{argmax}(Q[s_0, a]), \text{random}(a), \varepsilon), \forall a \in A$ 
    Estado_actual  $\leftarrow s \in S$ 
     $A_{\text{disponibles}} \leftarrow \text{AccionesDisponibles}(\text{Estado\_actual}, \text{Acción\_escogida})$ 
    while length(estado_actual)  $\neq$  num_acciones:
        Acción_escogida  $\leftarrow \varepsilon\text{-greedy}(\text{argmax}(Q[s_i, a]), \text{random}(a), \varepsilon),$ 
             $\forall a \in A_{\text{disponibles}}$ 
        Estado_actual  $\leftarrow s \in S$ 
         $A_{\text{disponibles}} \leftarrow \text{AccionesDisponibles}(\text{Estado\_actual}, \text{Acción\_escogida})$ 

    new_tree  $\leftarrow \text{ConstruirArbol}(\text{Estado\_actual})$ 
     $R \leftarrow -\text{pred}(\text{new\_tree})$ 
     $Q[s_i, a_i] \leftarrow R, \forall a_i, s_i \in \text{Estado\_actual}$ 
    /* Respecto a la línea anterior, recordar que el estado actual contiene información
    tanto del estado presente, de los estados visitados y de las acciones escogidas */
     $Q[s_{\text{final}}, a_{\text{final}}] \leftarrow \text{Función de valor según algoritmo Neo}, \forall a_{\text{final}} \in A, s_{\text{final}} \in S$ 

```


Para **Expected Sarsa**:

Hiperparámetros de entrada: $\alpha \in (0, 1), \varepsilon \in (0, 1), \gamma \in (0, 1), \text{num_iteraciones} \leftarrow 200$
 $\text{num_acciones} = \text{length}(A)$
 $Q[s, a] \leftarrow \text{zero_matrix}([\text{num_acciones}, \text{num_acciones}]), \forall s \in S, a \in A$
 $A_{\text{disponibles}} \leftarrow A, \forall a \in A$
for iteracion **in** num_iteraciones:
 Estado_actual $\leftarrow \text{empty_list}$
 Acción_escogida $\leftarrow \varepsilon\text{-greedy}(\text{argmax}(Q[s_0, a]), \text{random}(a), \varepsilon), \forall a \in A$
 Estado_actual $\leftarrow s \in S$
 $A_{\text{disponibles}} \leftarrow \text{AccionesDisponibles}(\text{Estado_actual}, \text{Acción_escogida})$
 while length(estado_actual) $\neq$ num_acciones:
 Acción_escogida $\leftarrow \varepsilon\text{-greedy}(\text{argmax}(Q[s_i, a]), \text{random}(a), \varepsilon),$
 $\forall a \in A_{\text{disponibles}}$
 Estado_actual $\leftarrow s \in S$
 $A_{\text{disponibles}} \leftarrow \text{AccionesDisponibles}(\text{Estado_actual}, \text{Acción_escogida})$
 new_subtree $\leftarrow \text{ConstruirSubarbol}(\text{Estado_actual})$
 $R \leftarrow -\text{pred}(\text{new_subtree})$
 $q_{\text{max}} \leftarrow \max(Q[s_i, a] \forall a \in A_{\text{disponibles}})$
 $\pi \leftarrow \varepsilon \frac{\text{ones_matrix}(\text{num_acciones})}{\text{num_acciones}} + (1 - \varepsilon) \frac{(Q[s_i, a] = q_{\text{max}})}{\text{sum}(Q[s_i, a])}, \forall a \in A_{\text{disponibles}}$
 $q_{\text{expected}} \leftarrow \pi \cdot \text{sum}(Q[s_i, a]), \forall a \in A_{\text{disponibles}}$
 $Q[s_{i-1}, a_{i-1}] \leftarrow Q[s_{i-1}, a_{i-1}] + \alpha(R + \gamma \cdot q_{\text{expected}} - Q[s_{i-1}, a_{i-1}])$

 $Q[s_{\text{final}}, a_{\text{final}}] \leftarrow Q[s_{\text{final}}, a_{\text{final}}] + \alpha(R - Q[s_{\text{final}}, a_{\text{final}}])$

REFERENCIAS BIBLIOGRÁFICAS

- [Ajanki, 2018] Ajanki, A. (2018). Differences between machine learning and software engineering.
<https://futurice.com/blog/differences-between-machine-learning-and-software-engineering>.
- [Amiri et al., 2018] Amiri, R., Mehrpouyan, H., Fridman, L., Mallik, R. K., Nallanathan, A., y Matolak, D. (2018). A machine learning approach for power allocation in hetnets considering qos. En *2018 IEEE International Conference on Communications (ICC)*, pp. 1–7. IEEE.
- [ApacheJena, 2020] ApacheJena (2020). Apache Jena.
<https://jena.apache.org/>.
- [Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684.
- [Buil-Aranda et al., 2013] Buil-Aranda, C., Hogan, A., Umbrich, J., y Vandenbussche, P.-Y. (2013). SPARQL web-querying infrastructure: Ready for action? En *International Semantic Web Conference*, pp. 277–293. Springer.
- [Buil-Aranda y Rojas, 2020] Buil-Aranda, C. y Rojas, C. (2020). Wikidata Benchmark.
<https://github.com/MillenniumDB/benchmark>.
- [Casals et al., 2020] Casals, D., Buil-Aranda, C., y Valle, C. (2020). Predicción de rendimiento de consultas SPARQL con *Deep Neural Network*.
- [Erling y Mikhailov, 2009] Erling, O. y Mikhailov, I. (2009). RDF support in the virtuoso DBMS. En *Networked Knowledge-Networked Media*, pp. 7–24. Springer.
- [François-Lavet et al., 2015] François-Lavet, V., Fonteneau, R., y Ernst, D. (2015). How to discount deep reinforcement learning: Towards new dynamic strategies. *arXiv preprint arXiv:1512.02011*.
- [Google, nd] Google (n.d). Machine learning crash course.
<https://developers.google.com/machine-learning/crash-course/ml-intro>.
- [Hasan y Gandon, 2014] Hasan, R. y Gandon, F. (2014). A machine learning approach to SPARQL query performance prediction. En *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volumen 1, pp. 266–273. IEEE.
- [Hüllermeier et al., 2013] Hüllermeier, E., Fober, T., y Mernberger, M. (2013). *Inductive Bias*, pp. 1018–1018. Springer New York, New York, NY.

- [LeCun et al., 2015] LeCun, Y., Bengio, Y., y Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.
- [Leis et al., 2015] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., y Neumann, T. (2015). How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215.
- [Marcus et al., 2019] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Pappamannouil, O., y Tatbul, N. (2019). Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*.
- [Medina, 2010] Medina, L. A. M. (2010). Query optimization using case-based reasoning, Chapter 2.
- [Mou et al., 2016] Mou, L., Li, G., Zhang, L., Wang, T., y Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. En *Thirtieth AAAI Conference on Artificial Intelligence*.
- [Murphy, 2012] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.
- [ontotext, nd] ontotext (n.d). What are linked data and linked open data?
<https://www.ontotext.com/knowledgehub/fundamentals/linked-data-linked-open-data/>.
- [OpenLinkSoftware, 2021] OpenLinkSoftware (1999-2021). Virtuoso OpenSource Documentation. <http://docs.openlinksw.com/virtuoso/index/>.
- [scalegrid.io, 2019] scalegrid.io (2019). 2019 database trends – SQL vs. NoSQL, top databases, single vs. multiple database use.
<https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/>.
- [Simeone, 2018] Simeone, O. (2018). A very brief introduction to machine learning with applications to communication systems. *IEEE Transactions on Cognitive Communications and Networking*, 4(4):648–664.
- [Socher et al., 2011] Socher, R., Huang, E., Pennington, J., Ng, A., y Manning, C. (2011). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. *Advances in Neural Information Processing Systems*, 24.
- [Sutton y Barto, 2018] Sutton, R. S. y Barto, A. G. (2018). *Reinforcement learning: An introduction, second edition*. MIT press.
- [Tai et al., 2015] Tai, K. S., Socher, R., y Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.

- [Tsialiamanis *et al.*, 2012] Tsialiamanis, P., Sidiourgos, L., Fundulaki, I., Christophides, V., y Boncz, P. (2012). Heuristics-based query optimisation for sparql. En *Proceedings of the 15th International Conference on Extending Database Technology*, pp. 324–335.
- [van Rijmenam, 2019] van Rijmenam, M. (2019). 7 steps to machine learning: How to prepare for an automated future.
<https://medium.com/dataseries/7-steps-to-machine-learning-how-to-prepare-for-an-automated-future-78c7918cb35d>.
- [W3C *et al.*, 2013] W3C, Harris, S., Seaborne, A., y Prud'hommeaux, E. (2013). SPARQL 1.1 query language. *W3C Recommendation*
<https://www.w3.org/TR/sparql11-query/>.
- [W3C *et al.*, 2014a] W3C, Klyne, G., Carroll, J. J., y McBride, B. (2014a). RDF 1.1 concepts and abstract syntax. *W3C Recommendation*
<https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [W3C *et al.*, 2014b] W3C, Manola, F., Miller, J., y McBride, B. (2014b). RDF 1.1 primer. *W3C Working Group*
<https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-primer/index.html>.
- [W3C *et al.*, 2007] W3C, Seaborne, A., y Hewlett-Packard Laboratories, B. (2007). The SPARQL Algebra.
<https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html>.
- [W3C y Zimmermann, 2014] W3C y Zimmermann, A. (2014). RDF 1.1: On semantics of RDF datasets. *W3C Recommendation*
<https://www.w3.org/TR/rdf11-datasets/>.