Métodos de Aprendizaje Automático para Grounding en Planning Clásico

por

Julia Olcese

Presentado ante la **FACULTAD DE MATEMÁTICA, ASTRONOMÍA, FÍSICA Y COMPUTACIÓN** como parte de los requerimientos para la obtención del grado de Licenciada en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CÓRDOBA

Marzo, 2025

Director: Dr. Carlos Areces

Resumen: La mayoría de los problemas de planificación se representan en PDDL, un lenguaje que permite definir acciones y predicados abstractos utilizando variables. Sin embargo, la mayoría de los planificadores automáticos encuentran planes basados en una representación completamente instanciada, por lo que es necesario asignar objetos concretos a dichas variables en un proceso conocido como grounding. Cuando el grounding genera todas las combinaciones posibles de instanciaciones, el número de acciones puede volverse excesivo, lo que resulta ineficiente para los algoritmos de búsqueda. Para abordar este problema, se propone un proceso de grounding más eficiente que, en lugar de instanciar todas las combinaciones posibles, utiliza modelos de traducción automática basados en la arquitectura Transformer para estimar la relevancia de cada acción. Luego, solamente las acciones más relevantes son instanciadas.

Palabras Claves: Planificación Automática, Aprendizaje automático, Transformer, Grounding, Modelos de Traducción Automática

Abstract: Most planning problems are represented in PDDL, a language that defines actions and predicates using variables. However, most automatic planners require fully instantiated representations, so it is necessary to replace the variables with objects through a process known as grounding. When grounding generates all possible instantiations, the resulting number of actions can become overwhelming, making search algorithms inefficient. To address this issue, we propose a more efficient grounding process that avoids instantiating all possible combinations and uses machine translation models based on the Transformer architecture to estimate the relevance of each action. Actions are then instantiated selectively based on this relevance.

Keywords: Automated Planning, Machine Learning, Transformer, Grounding, Machine Translation Models

Índice general

Reconocimientos

Este trabajo utilizó recursos computacionales de UNC Supercómputo (CCAD) de la Universidad Nacional de Córdoba[1], que forman parte del Sistema Nacional de Computación de Alto Desempeño (SNCAD) de la República Argentina. También se utilizó la computadora `atom` (FaMAF - UNC) adquirida con fondos de FaMAF y donaciones de AMD.

[1] https://supercomputo.unc.edu.ar

Agradecimientos

A Carlos, por guiarme y enseñarme con tanta paciencia a lo largo de este proceso. Al equipo de planning, por ayudarme en este trabajo.

A Pedro y Pedro, por ser parte de este tribunal y por ser excelentes profesores.

A Luis, Mari, Mati y Andrés por el aguante y la escucha de todos los días, y por acompañarme siempre, sea cual sea el camino que elija.

A Lala Lilia, Lolo Quito y Lala Stella, por estar presentes y atentos siempre, leyendo este trabajo, mandándome éxitos y preguntándome cómo me fue después de cada examen. Y a Lolo Luis, que le hubiese gustado acompañarme en este momento.

A mi familia, de sangre y postiza, por ser una red de contención y apoyo tan amorosa y valiosa.

A mis amigos de siempre, en especial a Agus, Ani, Emi, Jose y Nico, por tanto amor, juegos de mesa, risas, paciencia y compañía desde hace tantos años. Ojalá sea para siempre.

A Benja y Toni, por todos los trabajos y experiencias compartidas, y por el apoyo y amistad desde el primer día del cursillo. Agradecida de habernos encontrado.

A mis amigos de la facu, a cada persona con la que compartimos un caramelo preexamen, por hacer más llevaderos los largos días de clases y convertir este trayecto en algo mucho más lindo. En especial a Anna, Bruno, Lu, Lucas, Seba y Valen.

A cada profesor, ayudante alumno, y toda persona que haya contribuido a mi formación, en especial a Laura. Por cada clase, consulta, y duda resuelta dentro y fuera del aula, y por enseñar con tanta dedicación.

Y mi agradecimiento eterno a FaMAF, a la Universidad Nacional de Córdoba, y a la educación pública, gratuita, laica y de calidad. Espero que pueda seguir brindándole a otros las oportunidades que me dio a mí.

Capítulo 1

Introducción

1.1. Combinando planning y aprendizaje automático

En nuestro día a día, muchas situaciones requieren llevar a cabo instrucciones de forma ordenada para alcanzar un objetivo. Por ejemplo, para hacer una torta, debemos seguir una serie de pasos que podrían incluir prender el horno, mezclar los ingredientes, etc. Sin embargo, no todas las acciones pueden realizarse en cualquier momento; algunas dependen de que se cumplan ciertas condiciones previas. Por ejemplo, para poder meter la mezcla en el horno, primero es necesario que el horno esté prendido. Si el horno no está prendido, primero debemos realizar la acción de prender el horno y, después de eso, podemos meter la mezcla en el horno. Este ejemplo muestra cómo cada paso tiene precondiciones necesarias para poder realizarse y cómo cada acción produce efectos que modifican la situación actual.

La planificación automática (o planning) es un área de la inteligencia artificial que estudia este tipo de problemas: generar una secuencia de acciones (llamada plan) que permita alcanzar un objetivo deseado partiendo de un estado inicial, teniendo en cuenta las precondiciones necesarias para poder ejecutar cada acción y los efectos que producen una vez ejecutadas. Si definimos un estado como un conjunto de condiciones que se cumplen en un momento dado (como podría ser, 'el horno está prendido'), podemos pensar en las acciones como transformadoras de estados, ya que, al ejecutarse, modifican las condiciones existentes y generan un nuevo estado.

Es claro que para hacer un estudio matemático del problema de planning, es necesario utilizar una representación precisa; el problema debe ser descrito de manera formal, identificando qué acciones están disponibles, qué condiciones deben cumplirse para que una acción pueda ejecutarse, y qué cambios produce cada acción en el mundo. Sin embargo, no cualquier escenario puede ser modelado de esta forma; el planning es adecuado sólo para un conjunto restringido de situaciones.

Existen diversas aplicaciones prácticas del planning. Por ejemplo, en la gestión del tráfico urbano, se usa planning para coordinar los semáforos y minimizar embotellamientos (Vallati et al., 2016). También se usa en la gestión de riesgos empresariales, como en Sohrabi et al. (2019), donde se emplea planning para generar distintos escenarios posibles que permiten guiar la toma de decisiones. Otro ejemplo destacado es el sistema CASPER (Continuous Activity Scheduling Planning Execution and Replanning) (Knight et al., 2001), desarrollado por la NASA, que permite a las naves espaciales y rovers autónomos

ajustar dinámicamente sus actividades en tiempo real, evitando las demoras asociadas a la necesidad de enviar datos y recibir instrucciones desde la Tierra.

En la actualidad, los problemas de planning se suelen escribir en PDDL (Planning Domain Definition Language). Este lenguaje usa, en lugar de acciones y hechos concretos, esquemas de acciones y predicados con variables. Por ejemplo, en vez de enumerar acciones 'cortar tomate', 'cortar cebolla', 'cortar zanahoria', etc., en PDDL se define un esquema de acción 'cortar x', donde x es una variable que puede ser instanciada con objetos como 'tomate', 'cebolla', 'zanahoria', etc., obteniendo así todas las acciones posibles de un determinado tipo.

Para encontrar un plan, se utilizan algoritmos conocidos como planners. Es posible encontrar planes a partir de representaciones en PDDL, representaciones completamente instanciadas (es decir, sin variables), o representaciones semiinstanciadas. Sin embargo, las heurísticas de búsqueda más exitosas se basan en representaciones sin variables, por lo que la mayoría de los planners trabajan con hechos y acciones ya instanciados. Por lo tanto, si partimos de una representación en PDDL, es necesario instanciar las variables con objetos concretos para poder utilizar estos planners. Este proceso de instanciación se conoce como grounding.

En el grounding, la cantidad de acciones que se generan es exponencial en la cantidad de esquemas de acción y objetos con los que se pueden instanciar las variables. En problemas grandes, con una gran cantidad de acciones y objetos, se produce una enorme cantidad de acciones, y no todas son útiles para resolver el problema.

El problema de planning (presentado formalmente en la Sección 2.2) es decidible (Erol et al., 1995), lo que significa que hay algoritmos que para cualquier problema de planning van a terminar y decir si hay un plan que lo resuelva o no. Más aún, en caso que haya un plan que lo resuelva, es posible computarlo. Una forma intuitiva de demostrar esto es ver que si bien el número de planes posibles puede ser infinito (pues siempre podríamos seguir agregando acciones redundantes a un plan), el conjunto de posibles estados es finito. Esto es porque un estado es un conjunto de condiciones que se cumplen en un determinado momento, y si existen n condiciones posibles, cada una puede cumplirse o no en un estado dado, lo que da un total de 2^n estados posibles. Entonces, al tener una cantidad finita de estados, se puede explorar todas las secuencias de estados posibles, y determinar si alguna llega a un estado final en el que se incluyan las condiciones del objetivo.

Si bien siempre es posible decidir si un problema tiene un plan que lo resuelva o no (y encontrar el plan en caso de que exista), en problemas grandes, el tiempo necesario para explorar todas las combinaciones posibles de acciones puede volverse extenso. Para abordar el desafío que presentan los problemas grandes, hay dos estrategias principales: reducir el espacio de búsqueda (esto es, no probar todas las combinaciones posibles de acciones para formar planes) y reducir la cantidad de acciones con las que trabaja el planner.

Hay diversos trabajos que buscan formas de reducir el espacio de búsqueda, como por ejemplo en Hoffmann (2003). Estas propuestas consisten en definir heurísticas o reglas que guían al planner hacia combinaciones de acciones más prometedoras, evitando explorar todas las combinaciones posibles de acciones para formar planes.

Por otro lado, para reducir el conjunto de acciones disponibles para el planner, se pueden utilizar heurísticas para guiar el proceso de grounding. Esta estrategia no pasa

todas las acciones disponibles al planner, sino que descarta aquellas que probablemente no sean necesarias para resolver el problema.

Habiendo presentado brevemente el área de planning, ahora pasamos a introducir el aprendizaje automático, la segunda área fundamental para este trabajo.

El aprendizaje automático es una rama de la inteligencia artificial que se centra en desarrollar sistemas capaces de aprender y mejorar su desempeño a partir de datos, sin ser programados explícitamente. En lugar de seguir un conjunto fijo de reglas, un modelo de aprendizaje automático se entrena con ejemplos, detecta patrones en estos datos y utiliza esos patrones para tomar decisiones en situaciones nuevas. Este enfoque resulta especialmente útil en problemas donde sería impráctico definir manualmente todas las reglas posibles, o cuando ni siquiera se tiene en claro cuáles son las reglas a seguir. Por ejemplo, un sistema de recomendación de películas aprende a partir de las preferencias de los usuarios anteriores y, con base en esos patrones, sugiere nuevas películas que podrían gustarle a un usuario.

Esta tecnología se volvió muy popular en los últimos años, y muchas herramientas que usamos a diario utilizan aprendizaje automático, como el traductor de Google (y la mayoría de sistemas de traducción automática), el predictor de texto y ChatGPT. Sin embargo, una característica importante es que no siempre garantiza resultados correctos. Por ejemplo, ChatGPT puede generar respuestas incorrectas o imprecisas. A pesar de esto, el aprendizaje automático es una herramienta extremadamente flexible que puede aplicarse a una gran variedad de áreas y situaciones.

Una idea interesante es combinar el planning con el aprendizaje automático, aprovechar la flexibilidad del aprendizaje automático para intentar aprender información sobre los planes y explorar qué patrones y detalles pueden aportarnos. Sin embargo, al ser enfoques tan distintos, la combinación de estas dos estrategias no es trivial.

En este trabajo, buscamos usar aprendizaje automático para definir una heurística que guíe el grounding dependiendo de qué tan relevante es una acción para un problema específico. Dado el éxito de los modelos de traducción basados en aprendizaje automático (en particular aquellos con arquitectura Transformer), nuestro objetivo es ver si podemos aplicar algún algoritmo de traducción como heurística para orientar el grounding. Sin embargo, como ya se mencionó, los modelos de aprendizaje automático pueden cometer errores, con lo cual debemos usar con cuidado las salidas de los modelos, y manipular la información que nos brinda de forma tal que no se pierda la 'correctitud' en planning.

1.2. Estructura del trabajo

En el Capítulo 2, se presentan los conceptos teóricos fundamentales para este trabajo. Se abordan temas clave como la inteligencia artificial, la planificación automática, el aprendizaje automático y la arquitectura Transformer, que sirven como base para la propuesta de este trabajo.

En el Capítulo 3, se describe el problema principal y la solución propuesta. En este capítulo, se profundiza en el concepto de grounding y sus diversas variantes, así como la heurística propuesta para guiar el proceso de grounding, la cual se apoya en un modelo de traducción automática basado en la arquitectura Transformer. También se introduce una métrica denominada PUO, que permite evaluar la efectividad de la estrategia planteada.

El Capítulo 4 se centra en la implementación práctica y entrenamiento del modelo de traducción automática con arquitectura Transformer. Se presenta el código Python utilizado para construir el modelo, relacionando las distintas secciones con la teoría de esta arquitectura.

En el Capítulo 5, se presentan los resultados obtenidos a partir de una serie de experimentos diseñados para evaluar distintos aspectos de la solución propuesta.

Finalmente, en el Capítulo 6, se resumen las conclusiones derivadas de los resultados obtenidos y se plantean posibles trabajos futuros.

Capítulo 2

Marco teórico

En este capítulo se presentan los conceptos teóricos necesarios para el desarrollo del trabajo. Se introduce el concepto de *inteligencia artificial*, el campo dedicado a desarrollar sistemas capaces de realizar tareas que, tradicionalmente, requieren capacidades cognitivas humanas, como el razonamiento, el aprendizaje y la planificación.

Siguiendo esto último, se aborda el tema de *planificación automática* (*automatic planning*), un área que estudia la generación de secuencias de acciones que permiten alcanzar un objetivo determinado, partiendo de un estado inicial y siguiendo un conjunto de reglas. Esta área es fundamental en aplicaciones que requieren automatización de decisiones.

A continuación, se introduce el área de *aprendizaje automático* (*machine learning*), una subárea de la inteligencia artificial enfocada en construir sistemas que identifican y aprenden patrones directamente a partir de los datos disponibles, eliminando la necesidad de programar explícitamente el comportamiento del sistema. Al explicar aprendizaje automático, nos centramos en el *aprendizaje supervisado*, que es el utilizado en este trabajo.

Luego, se presentan las *redes neuronales artificiales*, modelos de aprendizaje automático inspirados en el funcionamiento de las redes neuronales biológicas. Estas redes son usadas como subcomponentes de modelos más avanzados de aprendizaje automático.

Se presenta también el área de *traducción automática*, que estudia cómo traducir texto entre idiomas de manera automática utilizando sistemas computacionales. Finalmente, se presenta la *arquitectura Transformer*, un modelo clave que ha revolucionado la tarea de traducción automática al permitir capturar dependencias globales entre los datos de entrada y salida.

2.1. Inteligencia artificial

La inteligencia artificial (IA) es un área de estudio que se enfoca en el análisis y construcción de sistemas inteligentes (Russell and Norvig, 2020). También puede definirse como la ciencia de hacer que las máquinas hagan cosas que requieren inteligencia al ser hechas por humanos (Minsky, 1968). A través del desarrollo de algoritmos que simulan procesos cognitivos humanos, la IA permite a las máquinas resolver problemas complejos, tomar decisiones autónomas y adaptarse a nuevas situaciones.

En la actualidad, la IA abarca una gran cantidad de subáreas, cada una enfocada en diferentes aspectos de la inteligencia humana. En este trabajo nos centramos en tres: la planificación automática, que genera secuencias de acciones para alcanzar determinados

objetivos; el aprendizaje automático, que permite a las máquinas aprender de los datos; y la traducción automática, que implica el proceso de convertir información de una representación a otra, como pasar de un lenguaje origen a un lenguaje destino. Otras áreas incluyen, por ejemplo, la visión por computadora, que permite a las máquinas interpretar imágenes y videos; la robótica, que busca desarrollar robots autónomos; y el razonamiento automático, que se enfoca en la toma de decisiones lógicas. Estas subáreas están interconectadas y contribuyen al avance de sistemas inteligentes en una variedad de campos.

A continuación, explicamos en detalle la planificación automática.

2.2. Planificación automática

Un área dentro de la inteligencia artificial es planificación automática (automatic planning, o simplemente planning). Planificar es el proceso de elegir y organizar acciones teniendo en cuenta los resultados que estas generan. En nuestro día a día, consciente o inconscientemente planificamos nuestras acciones de forma tal que nos permitan lograr algún objetivo determinado. Como dijimos que la inteligencia artificial busca estudiar los procesos que requieren inteligencia por parte de los humanos, es evidente que automatizar el proceso de planificación es un área de interés de la inteligencia artificial.

También hay una motivación práctica para el estudio de esta área: diseñar herramientas que permitan planificar de forma eficiente la resolución de una situación. Por ejemplo, podemos pensar en la organización de una operación de rescate luego de un desastre natural: se requieren varios actores y herramientas para la solución del problema, cuidadosamente planificadas y de manera urgente. Tareas de esta índole pueden ser organizadas utilizando planificadores automáticos (Ghallab et al., 2004).

Otra forma de pensar planning es como el enfoque basado en modelos para la selección de acciones en sistemas autónomos. A diferencia de otros enfoques en inteligencia artificial, como la programación explícita de reglas o el aprendizaje a partir de la experiencia, la planificación utiliza un modelo del entorno, las acciones y los objetivos para derivar automáticamente qué hacer a continuación (Geffner and Bonet, 2013).

A grandes rasgos, un problema de planning está dado por: un estado inicial, que representa las propiedades o condiciones que valen al comenzar el problema; una meta, que consiste en propiedades que se desea que valgan al terminar y un conjunto de acciones disponibles que modifican el estado.

Cada problema de planning está dado en un dominio determinado, que especifica reglas generales para todos los problemas que pertenecen a él. Por ejemplo, un dominio podría ser el de la planificación de vuelos de una aerolínea, donde se especifica que los objetos disponibles incluyen ‘Avión 1’, ‘Avión 2’, ‘Ezeiza’, ‘Ing. Ambrosio Taravella’; las propiedades relevantes podrían ser ‘El Avión X está en el Aeropuerto Y’ y ‘El Avión X tiene combustible’, y las acciones disponibles ‘Cargar combustible al Avión X’ o ‘Volar Avión X de A a B’.

Dentro de este dominio, se pueden plantear diferentes problemas de planificación, cada uno con su propio estado inicial, meta y objetos dispoibles, dentro de las reglas definidas por el dominio. Un problema podría tener como estado inicial ‘El Avión 1 está en el Aeropuerto Ezeiza’, y la meta ‘El Avión 1 está en el Aeropuerto Ing. Ambrosio Taravella’. Otro problema podría ser el dado por el estado inicial ‘El Avión 2 está en el Aeropuerto Ezeiza’ y ‘El Avión 2 tiene combustible’, y la meta ‘El Avión 1 está en el

Aeropuerto Ing. Ambrosio Taravella' y 'El Avión 2 tiene combustible". Ambos problemas trabajan con las mismas acciones y propiedades, pero establecen situaciones distintas, que se resuelven de maneras diferentes.

Así, dado un problema de planning en un determinado dominio, un planificador automático busca devolver una secuencia ordenada de acciones que permitan llegar del estado inicial a un estado final que satisfaga la meta. Esta secuencia es llamada un plan que resuelve el problema.

A continuación se presentan dos lenguajes utilizados para representar problemas de planning, estos son *STRIPS* y *PDDL*.

2.2.1. STRIPS

STRIPS (STanford Research Institute Problem Solver) es un planificador automático presentado en Fikes and Nilsson (1971). La representación de los problemas utilizada por este planificador, conocida como el lenguaje STRIPS, se convirtió en un estándar ampliamente usado en la planificación automática debido a su simplicidad y eficacia para modelar dominios de planificación. A continuación, se analiza su estructura y características principales.

Formalmente, un problema de planning en el lenguaje STRIPS es una 4-upla $\Pi = \langle F, O, I, G \rangle$, donde F es un conjunto de símbolos proposicionales llamados facts o hechos, O es un conjunto de operadores, I es un subconjunto de F llamado estado inicial y G es un subconjunto de F llamado objetivo o meta.

Un estado entonces es un conjunto de símbolos proposicionales $s \subseteq F$ que son verdaderos en un determinado momento. Sea s un estado, y sea $f \in F$ tal que $f \notin s$, decimos que el hecho f no se cumple en el estado s.

Los operadores son la representación de las acciones en el lenguaje STRIPS. Dado un problema $\Pi = \langle F, O, I, G \rangle$, un operador $o \in O$ es una 3-upla $o = \langle pre(o), add(o), del(o) \rangle$ donde $pre(o)$, $add(o)$ y $del(o)$ son subconjuntos de F, y los llamamos precondiciones de o, lista a agregar de o, y lista a eliminar de o respectivamente. Los hechos que están en $pre(o)$ son aquellos que deben cumplirse para que se pueda realizar la acción. Por otro lado, $add(o)$ y $del(o)$ representan la postcondición que se cumple al ejecutar la acción: los hechos de $add(o)$ son los que se cumplirán gracias a la ejecución de la acción o; y los de $del(o)$ son los que se dejan de cumplir una vez que se ejecuta la acción o.

Un operador o se dice aplicable en un estado s si $pre(o) \subseteq s$. Si el operador o es aplicable en el estado s, al aplicarlo se obtiene un nuevo estado s' que resulta de eliminar de s los símbolos proposicionales que aparecen en $del(o)$, y agregar los que aparecen en $add(o)$. Esto es, $s' = (s \setminus del(o)) \cup add(o)$. En este caso, denotamos $s \xrightarrow{o} s'$ a la transición del estado s al s' por medio de la aplicación del operador o. Podemos pensar entonces en los operadores como transformadores de estados, eliminando y agregando hechos al estado en el que se aplican.

Dada una secuencia de operadores $\bar{o} = [o_1, o_2, \ldots, o_n]$ y estados $s_0, s_1, \ldots, s_n$ tales que $s_0 \xrightarrow{o_1} s_1$, $s_1 \xrightarrow{o_2} s_2$, $\ldots$, $s_{n-1} \xrightarrow{o_n} s_n$, escribimos $s_o \xrightarrow{\bar{o}} s_n$ para expresar que los operadores se pueden aplicar iterativamente para llegar del estado s_0 al estado s_n.

Así, dado un problema $\Pi = \langle F, O, I, G \rangle$, el algoritmo de STRIPS busca una secuencia ordenada de operadores tales que permitan transformar el estado desde I hasta un estado final que incluya a la meta G. Es decir, busca $\bar{o}$ tal que $I \xrightarrow{\bar{o}} sG$ donde $G \subseteq sG$. En el caso

de existir, $\bar{o}$ se denomina plan de Π.

A continuación se presenta en lenguaje STRIPS un problema en el dominio de la aerolínea descripto anteriormente:

$$F = \{Aterrizado(Avión1, Ezeiza),\ Aterrizado(Avión2, Ezeiza),\\ Aterrizado(Avión1, IngAT),\ Aterrizado(Avión2, IngAT),\\ TieneCombustible(Avión1),\ TieneCombustible(Avión2)\}$$

$$O = \{Volar(Avión1, Ezeiza, IngAT),\ Volar(Avión1, IngAT, Ezeiza),\\ Volar(Avión2, Ezeiza, IngAT),\ Volar(Avión2, IngAT, Ezeiza),\\ CargarCombustible(Avión1),\ CargarCombustible(Avión2)\}$$

$$I = \{Aterrizado(Avión1, Ezeiza),\ Aterrizado(Avión2, Ezeiza)\}$$

$$G = \{Aterrizado(Avión1, IngAT),\ TieneCombustible(Avión1)\}$$

con los operadores de O de la forma

$$Volar(A, Or, Des) = \langle \\ pre : \{Aterrizado(A, Or),\ TieneCombustible(A)\},\\ add : \{Aterrizado(A, Des)\},\\ del : \{Aterrizado(A, Or),\ TieneCombustible(A)\}\\ \rangle$$

$$CargarCombustible(A) = \langle \\ pre : \emptyset,\\ add : \{TieneCombustible(A)\},\\ del : \emptyset\\ \rangle$$

En este problema, en el estado inicial tenemos que tanto el Avión 1 como el Avión 2 están aterrizados en Ezeiza, y queremos llegar a un estado final en el que el Avión 1 esté aterrizado en Ingeniero Ambrosio Taravella y tenga combustible. Un posible plan para este problema es la siguiente secuencia de acciones:

[*CargarCombustible(Avión1)*, *Volar(Avión1, Ezeiza, IngAT)*, *CargarCombustible(Avión1)*].

Es importante destacar que en este ejemplo se esquematizaron los operadores de O para simplificar la representación. Se usó A como una variable para representar un avión cualquiera, y Or y Des como variables para representar aeropuertos cualesquiera. El proceso de reemplazar estas variables en los esquemas con aeropuertos o aviones concretos para así obtener un operador se conoce como *grounding*. En un problema de STRIPS formal, los operadores se enumeran exhaustivamente detallando las precondiciones, lista

a agregar y lista a eliminar de cada uno de ellos, así O formalmente tendría la siguiente forma:

$$
\begin{aligned}
O = \{ & \\
& \langle \\
& \{Aterrizado(Avión1, Ezeiza),\ TieneCombustible(Avión1)\}, \\
& \{Aterrizado(Avión1, IngAT)\}, \\
& \{Aterrizado(Avión1, Ezeiza),\ TieneCombustible(Avión1)\} \\
& \rangle, \\
& \langle \\
& \{Aterrizado(Avión1, IngAT),\ TieneCombustible(Avión1)\}, \\
& \{Aterrizado(Avión1, Ezeiza)\}, \\
& \{Aterrizado(Avión1, IngAT),\ TieneCombustible(Avión1)\} \\
& \rangle, \\
& \ldots \\
\} &
\end{aligned}
$$

Es evidente que enumerar exhaustivamente todos los operadores sería inviable en problemas grandes. Es por esto que aparece otro lenguaje usado para representar problemas de planning, que sigue esta idea de no enumerar todos los operadores posibles, sino esquematizarlos usando variables. Este lenguaje se presenta a continuación.

2.2.2. PDDL

Otra forma de representar los problemas de planning es usando un lenguaje llamado Planning Domain Definition Language (PDDL), presentado en McDermott et al. (1998). Este lenguaje permite la especificación de dominios y problemas de planning usando variables, lo que facilita la representación de problemas de gran tamaño.

Formalmente, podemos pensar un problema de planning en PDDL como una 6-upla $\Pi = \langle P, A, \Sigma^C, \Sigma^O, I, G \rangle$ donde P es un conjunto de predicados, A es un conjunto de esquemas de acciones, Σ^C es un conjunto de objetos constantes, Σ^O es un conjunto de objetos variables, I y G son conjuntos de símbolos proposicionales que representan el estado inicial y el objetivo respectivamente.

A diferencia de la representación STRIPS, en PDDL no se define un conjunto de hechos, sino un conjunto de predicados P. Estos predicados pueden incluir variables que pertenecen a Σ^C, y al instanciarlas con elementos de Σ^O, se obtienen símbolos proposicionales concretos, es decir, hechos. Así, denotamos a los predicados de P de la forma $p[X]$, donde $X \subseteq \Sigma^O$ es el conjunto de variables que ocurren en el predicado, y lo llamamos interfaz de p.

Por ejemplo, si consideramos el dominio de la aerolínea, podríamos tener el predicado $tieneCombustible(avión)[avión]$, donde $avión \in \Sigma^C$ es una variable que luego puede ser reemplazada por ejemplo por $Avión2 \in \Sigma^O$ y obtener el símbolo proposicional $tieneCombustible(Avión2)$.

Denotamos P^{Σ^C} al conjunto de hechos que se obtiene como resultado de instanciar los predicados de P con todas las combinaciones posibles de constantes de Σ^C. Tanto el estado inicial I como el objetivo G son subconjuntos de P^{Σ^C}.

Además, en lugar de tener operadores, en PDDL se definen esquemas de acciones, los cuales también permiten variables. Así, un esquema de acción de A se representa como $a[X]$ y es una terna de la forma $a[X] = \langle pre(a), add(a), del(a) \rangle$, donde $pre(a)$, $add(a)$ y $del(a)$ son subconjuntos de P, posiblemente preinstanciados. Llamamos interfaz de a al conjunto $X \subseteq \Sigma^O$ de variables que aparecen en $pre(a) \cup add(a) \cup del(a)$. Llamamos a $pre(a)$ precondiciones de a, $add(a)$ lista a agregar de a, y $del(a)$ lista a eliminar de a.

Por ejemplo, volviendo al dominio de la aerolínea, un posible esquema de acción es $CargarCombustible(avión)[avión] = \langle \emptyset, tieneCombustible(avión), \emptyset \rangle$, donde $avión \in \Sigma^C$ es una variable que al ser instanciada, por ejemplo, con $Avión2 \in \Sigma^O$ genera el operador $CargarCombustible(Avión2) = \langle \emptyset, tieneCombustible(Avión2), \emptyset \rangle$.

Notemos que un problema PDDL se puede separar en la especificación del dominio (P, A, Σ^C) (que va a ser la misma para todos los problemas del dominio), y la especificación del problema (Σ^O, I, G) (que varía para cada problema). Así, para representar un problema en el lenguaje PDDL, se utiliza un archivo para la especificación del dominio y otro para la definición del problema.

Consideremos nuevamente el ejemplo de dominio de la aerolínea, se presenta en 2.1 el código usado en el archivo de especificación del dominio en PDDL, y en 2.2 el código correspondiente al archivo de la especificación del problema.

Código 2.1: Dominio Aerolínea en PDDL

```
(
define (domain AEROLINEA-DOMAIN)

(:requirements :strips :typing)

(:types AVION AEROPUERTO)

(:predicates (aterrizado ?x - AVION ?y - AEROPUERTO)
             (tieneCombustible ?x - AVION))

(:action volar
:parameters (?x - AVION ?o - AEROPUERTO ?d - AEROPUERTO)
:precondition (and (aterrizado ?x ?o)
                   (tieneCombustible ?x))
:effect (and (aterrizado ?x ?d)
             (not (aterrizado ?x ?o))
             (not (tieneCombustible ?x)))
)

(:action cargarCombustible
:parameters (?x - AVION)
:precondition ()
:effect (tieneCombustible ?x)
)
)
```

Código 2.2: Problema en Dominio Aerolínea en PDDL

```
(
define (problem AEROLINEA-TASK)

(:domain AEROLINEA-DOMAIN)

(:objects avion1 avion2 - AVION ezeiza ingAT - AEROPUERTO)

(:init (and (aterrizado avion1 ezeiza)
            (aterrizado avion2 ezeiza)
            (tieneCombustible avion1)))

(:goal (and (aterrizado avion1 ingAT) (tieneCombustible avion1)))
)
```

En el formalismo tenemos:

$$P = \{Aterrizado(avión, aero),\ TieneCombustible(avión)\}$$

$$A = \{Volar(avión, aero, aero'),\ CargarCombustible(avión)\}$$

$$\Sigma^C = \{Avión1,\ Avión2,\ Ezeiza,\ IngAT\}$$

$$\Sigma^O = \{avión,\ aero,\ aero'\}$$

$$I = \{Aterrizado(Avión1, Ezeiza),\ Aterrizado(Avión2, Ezeiza)\}$$

$$G = \{Aterrizado(Avión1, IngAT),\ TieneCombustible(Avión1)\}$$

con

$$\begin{aligned} Volar(avión, aero, aero') = \langle & \\ & pre : \{Aterrizado(avión, aero),\ TieneCombustible(avión)\}, \\ & add : \{Aterrizado(avión, aero')\}, \\ & del : \{Aterrizado(avión, aero),\ TieneCombustible(avión)\} \\ & \rangle \end{aligned}$$

$$\begin{aligned} CargarCombustible(avión) = \langle & \\ & pre : \emptyset, \\ & add : \{TieneCombustible(avión)\}, \\ & del : \emptyset \\ & \rangle \end{aligned}$$

La representación con PDDL es ámpliamente utilizada debido a su capacidad para expresar problemas de planificación de manera concisa y estructurada. Sin embargo, la mayoría de los planners automáticos trabajan con representaciones STRIPS de un problema ya que las heurísticas más eficientes para la búsqueda de planes se basan en representaciones sin variables.

Así, uno de los desafíos clave en la planificación automática es el proceso de grounding, que permite pasar de representaciones abstractas con variables a instancias concretas que los algoritmos de planificación pueden procesar. Es importante destacar que dado un esquema de acción, la cantidad de operadores concretos que se pueden obtener instanciando el esquema crece exponencialmente en función de la cantidad de variables del esquema, y los objetos disponibles para instanciar cada una de ellas. En muchos casos, el resultado del grounding puede ser tan extenso que no cabe en la memoria, con lo cual el planificador no llega a procesar siquiera el problema. Queda en evidencia entonces la importancia del grounding en el proceso de planning: es fundamental reducir la cantidad de acciones instanciadas sin comprometer la capacidad de encontrar soluciones viables.

En este contexto, las técnicas de aprendizaje automático ofrecen un enfoque prometedor para superar las limitaciones del grounding: al predecir la relevancia de las acciones instanciadas se pueden descartar acciones innecesarias para la resolución del problema, y así reducir el tamaño del problema. A continuación se da una introducción al área de aprendizaje automático, para luego poder analizar su uso en el contexto de grounding.

2.3. Aprendizaje automático

El aprendizaje automático (machine learning) es una subárea de la inteligencia artificial que estudia algoritmos que mejoran su rendimiento a partir de la experiencia, en cuyo caso decimos que ha "aprendido". Esta mejora se mide en función de una métrica predefinida, específica para el problema que el programa está diseñado para resolver (Mitchell, 1997).

Por ejemplo, si queremos construir un programa que clasifique correos electrónicos como spam o no spam, la experiencia de la que el programa va a aprender consiste en un conjunto de correos electrónicos ya etiquetados como spam o no spam (conjunto de entrenamiento) y la métrica para medir el rendimiento podría ser la cantidad de correos clasificados correctamente. Inicialmente, el programa no sabe cómo clasificar un correo, pero a medida que analiza los ejemplos en el conjunto de entrenamiento, comienza a identificar patrones comunes en los correos de spam y no spam. De este modo, cuando se le presenta un correo nuevo, el programa es capaz de predecir su clasificación basándose en los patrones que aprendió de los ejemplos etiquetados. Podemos pensar en una función f tal que cuando le damos un correo como entrada, la salida nos dice si es spam o no, y nuestro objetivo es aproximar esa función basándonos en algunos puntos conocidos que serían el conjunto de entrenamiento. Al programa que aproxima la función se lo denomina modelo.

Una característica fundamental de este proceso es que el programa no recibe instrucciones explícitas sobre los patrones que debe buscar. No se le ha dicho directamente cómo detectar el spam; en cambio, aprende a hacerlo de manera autónoma al procesar y analizar los datos del conjunto de entrenamiento. Este aprendizaje basado en datos permite que el programa generalice, es decir, que pueda clasificar correctamente correos electrónicos

que nunca ha visto, aplicando los patrones identificados en el conjunto de entrenamiento a situaciones nuevas.

La inferencia es el proceso mediante el cual un modelo, una vez entrenado, utiliza lo que ha aprendido para hacer predicciones sobre nuevos datos. Durante esta etapa, el modelo toma un conjunto de datos de entrada (por ejemplo, un correo electrónico no visto previamente) y calcula una salida (por ejemplo, la clasificación como spam o no spam) basándose en los patrones y relaciones que ha identificado durante el entrenamiento. La inferencia es entonces la aplicación del modelo a nuevos ejemplos para obtener respuestas o predicciones una vez que ya terminó la etapa de aprendizaje.

Hay distintas formas en las cuales un programa puede aprender. Se clasifican en:

1. Aprendizaje supervisado: aprende a partir de un conjunto de entrenamiento, el cual tiene datos etiquetados (esto es con la respuesta correcta). Es el ejemplo de clasificación de correos y es el que se utiliza en este trabajo.

2. Aprendizaje no supervisado: los datos del conjunto de entrenamiento no están etiquetados, el programa busca agrupar los datos del conjunto según patrones comunes que encuentre entre ellos.

3. Aprendizaje por refuerzo: es un punto medio entre aprendizaje supervisado y no supervisado. Se saben las etiquetas de los datos, pero cuando el programa se equivoca, se le informa que la respuesta no es correcta pero no se le dice cuál debería ser la respuesta.

2.3.1. Aprendizaje supervisado

En este trabajo nos vamos a centrar en el aprendizaje supervisado. Como se mencionó anteriormente, tenemos un conjunto de datos etiquetados (llamado conjunto de entrenamiento), donde cada elemento del conjunto es un par (x_i, t_i), siendo x_i una entrada y t_i la respuesta correcta asociada a esta entrada. El objetivo es construir un modelo que sea capaz de aproximar una función y, sabiendo que $y(x_i) = t_i$ para cada dato de entrenamiento.

Consideremos un ejemplo sencillo: Tenemos como conjunto de entrenamiento los pares $\{(1,5),(2,10),(4,20),(100,500),\dots\}$. Para construir el modelo, primero es necesario definir su arquitectura, es decir cómo va a calcular la salida dada una entrada. Podríamos proponer una arquitectura de la forma $M(x;\mathbf{p}) = a \cdot x + b$, donde $\mathbf{p} = [a,b]$ son parámetros de la arquitectura y x la entrada que se le va a pasar. Una vez definida la arquitectura, se realiza el proceso de entrenamiento, en el que se le van presentando elementos del conjunto de entrenamiento al modelo de forma tal que pueda ajustar el valor de sus parámetros (en este caso, a y b) según la salida esperada. Al terminar el proceso de entrenamiento, idealmente tendríamos que el modelo aprendió que a debe tomar el valor 5 y b debe tomar el valor 0. Denotamos entonces $M_{\mathbf{p}^*}(x) = 5 \cdot x + 0$ con $\mathbf{p}^* = [5,0]$, la función resultante de reemplazar los parámetros a y b por 5 y 0 respectivamente, siendo este es el modelo final.

A continuación se detalla el proceso de entrenamiento de un modelo de aprendizaje automático.

Una vez que se estableció la arquitectura $M(\mathbf{x};\mathbf{p})$ del modelo, los parámetros del mismo se inicializan en valores arbitrarios. Esto es, comenzamos trabajando con $M_{\mathbf{p}^*}(\mathbf{x})$ donde $\mathbf{p}$ tiene valores aleatorios. Durante el proceso de entrenamiento, se presentan iterativamente ejemplos al modelo y se ajustan los valores de $\mathbf{p}$ según la distancia entre la salida

obtenida al pasar la entrada por el modelo y la salida esperada, dada por el conjunto de entrenamiento. Para medir esta distancia, se utiliza una función de pérdida. Hay diversas funciones de pérdida utilizadas, una de las más conocidas y clásicas es el *Error Cuadrático Medio*. Sea $M(\mathbf{x};\mathbf{p})$ una arquitectura, $\mathbf{p}^*$ una asignación de valores a sus parámetros, y $\{(x_i,y_i),...,(x_n,y_n)\}$ el conjunto de entrenamiento; el error cuadrático medio para $M_{\mathbf{p}^*}$ está dado por

$$L(\mathbf{p}^*) = \frac{1}{2n}\sum_{i=1}^{n}(M_{\mathbf{p}^*}(x_i) - y_i)^2 \tag{2.1}$$

Notar que si $M_{\mathbf{p}^*}$ predice de forma correcta todas las salidas, la pérdida sería 0, pues tendríamos $M_{\mathbf{p}^*}(x_i) = y_i$ para todo i. Así, el objetivo del entrenamiento es encontrar valores para $\mathbf{p}^*$ que permitan minimizar la pérdida.

Una vez calculada la pérdida, para actualizar los valores de $\mathbf{p}^*$, se calcula el gradiente de la pérdida en $\mathbf{p}^*$, denotado $\nabla L(\mathbf{p}^*)$, que señala la dirección en la que la función de pérdida aumenta más rápidamente con respecto a cada parámetro del modelo. Luego, los parámetros se actualizan mediante un algoritmo de optimización, como el descenso de gradiente, en el cual los pesos se actualizan siguiendo la dirección opuesta al gradiente:

$$\mathbf{p}^* = \mathbf{p}^* - \eta \cdot \nabla L(\mathbf{p}^*) \tag{2.2}$$

Donde η es la tasa de aprendizaje, que determina la "intensidad" de las actualizaciones de los parámetros. Este valor debe elegirse cuidadosamente, ya que si es demasiado pequeño, las actualizaciones serán lentas y el proceso de aprendizaje podría ser ineficiente. Por otro lado, si es demasiado grande, las actualizaciones podrían ser demasiado drásticas, lo que podría hacer que el modelo no converja al mínimo deseado.

La actualización de pesos puede realizarse después de cada ejemplo que se presenta a la red, o bien se pueden agrupar varios ejemplos en un lote (batch), y hacer la actualización luego de procesar todos los ejemplos del batch. En este caso, es necesario definir el tamaño del lote (batch size).

Un pase completo por todos los datos de entrenamiento se llama época (epoch). Para el proceso de entrenamiento se determina un número de épocas, es decir, cuántas veces se le presentan a la red todos los ejemplos de entrenamiento.

Entonces, el proceso de entrenamiento consiste en repetir durante una cantidad predefinida de épocas, para cada lote, los siguientes pasos:

- Presentar todos los ejemplos de entrenamiento del lote al modelo (o un solo ejemplo, si el lote tiene tamaño 1).
- Calcular la pérdida para el/los ejemplo/s presentado/s.
- Actualizar los parámetros usando un optimizador.

En el ejemplo presentado en esta sección, se trabajó con una arquitectura simple de aprendizaje automático dada por $M(x;\mathbf{p}) = a \cdot x + b$, la cual fue útil para capturar las relaciones presentes en el ejemplo. Sin embargo, esta es una arquitectura bastante sencilla y, en la mayoría de casos del mundo real, los datos no siguen una relación lineal tan directa. En estos escenarios, se requieren modelos capaces de capturar relaciones más

complejas entre los datos, y es en este contexto que entran en juego las redes neuronales artificiales.

Las redes neuronales artificiales son una arquitectura de aprendizaje automático que puede aprender patrones más complejos en los datos, lo que las convierte en una herramienta fundamental para tareas como la traducción de idiomas, la clasificación de imágenes, entre otras. En la siguiente sección, exploraremos los aspectos fundamentales de las redes neuronales artificiales y su funcionamiento.

2.4. Red neuronal artificial

Una red neuronal artificial es una arquitectura de aprendizaje automático inspirada en las redes neuronales biológicas. Una red neuronal está formada por un conjunto de unidades de cómputo llamadas neuronas, organizadas en capas. Un precedente importante para el desarrollo de esta arquitectura es el trabajo de McCulloch and Pitts (1943), en el que se modela matemáticamente el funcionamiento de las neuronas biológicas.

Una categoría en particular de redes neuronales son las llamadas *feed forward*, en las cuales las capas se organizan de manera sucesiva y cada capa recibe información solamente de la capa anterior, y envía información solamente a la capa siguiente.

La primera capa es llamada capa de entrada, la cual no es una capa de neuronas sino el punto donde se envían los datos de entrada. La última capa es llamada capa de salida, que es donde se obtiene la predicción final del modelo. Las capas que se encuentran entre la de entrada y la de salida se conocen como capas ocultas. Un tipo de redes neuronales son las *fully connected*, donde todas las neuronas de una capa envían información a cada una de las neuronas de la capa siguiente.

En la Figura 2.1 se muestra un ejemplo de una red neuronal feed forward fully connected, la cual tiene una capa de entrada que recibe los datos (dos valores de entrada); una primera capa oculta con cuatro neuronas; una segunda capa oculta con tres neuronas; y una capa de salida con una única neurona. Las flechas representan el envío de información desde la neurona de la izquierda a la de la derecha, y podemos observar que la información siempre se envía solamente a la capa siguiente (feed forward), y cada neurona recibe información de todas las neuronas de la capa anterior (fully connected).

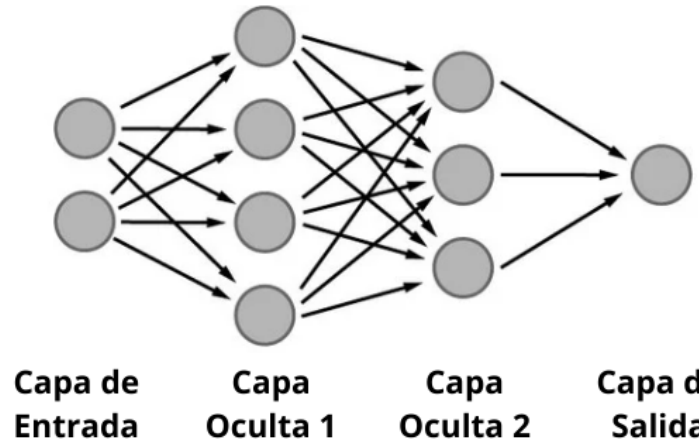


Figura 2.1: Ejemplo de red neuronal feed forward fully connected

Se estudia a continuación el funcionamiento de una red neuronal artificial feed forward fully connected, que será de utilidad en este trabajo.

Cada neurona recibe una entrada de cada una de las n neuronas de la capa anterior (o los n datos de entrada en caso de que la capa anterior sea la capa de entrada), y hace una suma ponderada de las mismas:

$$z = \sum_{i=1}^{n} w_i x_i = \mathbf{w}' \cdot \mathbf{x}' \tag{2.3}$$

donde $\mathbf{x}' = [x_1, ..., x_n]$ es la entrada y $\mathbf{w}' = [w_1, ..., w_n]$ son parámetros llamados pesos sinápticos.

Siguiendo las redes neuronales biológicas, si esta suma supera un umbral μ, la neurona dispara. Para modelar este umbral, se agrega una entrada adicional con valor fijo -1, y el peso asociado a esta entrada corresponde al valor del umbral. Así, la suma ponderada queda:

$$z = \sum_{i=1}^{n} w_i x_i - \mu = \sum_{i=1}^{n+1} w_i x_i = \mathbf{w} \cdot \mathbf{x} \tag{2.4}$$

con $\mathbf{w} = [w_1, ..., w_n, \mu]$ y $\mathbf{x} = [x_1, ..., x_n, -1]$.

Esta suma ponderada luego pasa por una función de activación. Existen varias funciones de activación, pero una de las más populares es la función ReLU (Rectified Linear Unit), dada por:

$$ReLU(x) = \text{máx}(0, x) \tag{2.5}$$

Las funciones de activación juegan un papel crucial en las redes neuronales, ya que introducen no linealidad en el modelo. Sin la no linealidad, no importaría cuántas capas tenga la red, ya que, matemáticamente, una sucesión de combinaciones lineales de las entradas siempre se puede reducir a una única combinación lineal. De esta manera, sin una función de activación no sería posible aprender patrones complejos y la red quedaría limitada a una simple transformación lineal de la entrada.

Cada neurona tiene sus propios pesos sinápticos asociados, que determinan la influencia de cada dato de entrada en el cálculo de su salida. Además, cada capa de la red tiene su propia función de activación específica. Supongamos que tenemos una capa de n neuronas, que recibe un vector de entrada de tamaño m, representamos los pesos sinápticos de la capa como una matriz $W \in \mathbb{R}^{n \times m}$, donde W_{ij} corresponde al peso sináptico de la i-ésima neurona para el j-ésimo dato de entrada. Así, sea W la matriz de pesos de una determinada capa, g su función de activación y $\mathbf{x}$ su entrada, la salida de la capa está dada por $g(W \cdot \mathbf{x})$.

Cuando la red recibe una entrada, esta pasa por todas las neuronas de la primera capa oculta, y cada una de esas neuronas genera una salida. Las salidas de todas las neuronas forman un nuevo vector, que se convierte en la entrada para la siguiente capa. Este proceso se repite capa por capa: las salidas de una capa se convierten en las entradas de la siguiente. Las neuronas finales producen los resultados de la red, que están basados en los cálculos realizados en todas las capas anteriores.

Así, la red completa puede entenderse como una composición de funciones. Supongamos una red neuronal de $n+1$ capas (donde la capa 0 es la capa de entrada, y la capa n es la capa de salida), con g_i y W^i la función de activación y matriz de pesos de la i-ésima

capa respectivamente, y sea $\mathbf{x}$ la entrada de la red, la salida de la red está dada por:

$$o = g_{n+1}(W^{n+1} \cdot g_n(W^n \cdot (...g_1(W^1 \cdot \mathbf{x})...))) \tag{2.6}$$

Al definir la arquitectura de una red neuronal, se establecen de antemano la cantidad de capas, y para cada una de ellas la cantidad de neuronas y la función de activación. Estos son llamados hiperparámetros de la red, y se mantienen fijos. Durante el proceso de entrenamiento, se buscan encontrar los valores "óptimos" de los pesos, óptimos en el sentido de que permitan a la red generalizar y poder predecir de manera correcta el resultado de nuevos ejemplos. Llamamos a los pesos sinápticos parámetros de la red, y se distinguen de los hiperparámetros justamente porque se van a ir ajustando a medida que se entrena.

Para entrenar una red neuronal, se sigue un procedimiento similar al presentado en la Sección 2.3.1: se presentan ejemplos de entrenamiento a la red, se calcula la salida de la red para cada ejemplo mediante un proceso conocido como *forward propagation* (ya que la información fluye en la red desde la entrada hasta la salida), luego se mide el error entre la salida obtenida y la esperada mediante una función de pérdida equivalente a 2.2 y finalmente se actualizan los pesos de la red mediante el descenso por el gradiente. Este último paso requiere calcular las derivadas parciales de la función de pérdida con respecto a cada parámetro de la red para actualizar los pesos en la dirección que minimiza el error.

Dado que la salida de la red es una composición de funciones, para calcular las derivadas parciales será necesario usar la regla de la cadena. Sin embargo, muchas derivadas parciales se reutilizan a lo largo del proceso, por lo que recalcularlas en cada paso sería ineficiente. Para optimizar este proceso, se hace uso de un algoritmo conocido como *backpropagation*, que propaga el error en sentido inverso, desde la salida hasta la entrada, almacenando y reutilizando las derivadas parciales intermedias en lugar de recalcularlas (Goodfellow et al., 2016).

Las redes neuronales artificiales son herramientas poderosas para modelar relaciones complejas en los datos. Es por esto que se utilizan como subcomponentes en diversas arquitecturas de aprendizaje automático para distintas tareas. En el caso de la traducción automática, los modelos basados en la arquitectura Transformer incorporan redes neuronales feed forward fully connected como parte de su estructura. A continuación, se introduce el área de traducción automática, seguido de una explicación detallada sobre la arquitectura Transformer.

2.5. Traducción automática

La traducción automática es el proceso de utilizar algoritmos y modelos computacionales para traducir texto o discurso de un idioma a otro sin intervención humana directa. El objetivo es que una máquina pueda tomar una secuencia de palabras en un idioma (el idioma de origen) y generar una secuencia equivalente en otro idioma (el idioma de destino). Si bien comúnmente se usa para traducir entre lenguajes humanos (como español o inglés), también puede usarse para convertir información entre otros tipos de sistemas de representación. Por ejemplo, el modelo TransCoder, desarrollado por Facebook AI Research, permite traducir código entre lenguajes de programación como Python, C++ y Java (Lachaux et al., 2020).

El aprendizaje supervisado es un enfoque comúnmente utilizado para la traducción automática. En este enfoque, el modelo se entrena con pares de frases en el idioma de origen y su traducción correspondiente en el idioma de destino. Así, en el proceso de entrenamiento, el modelo aprende patrones lingüísticos y estructuras gramaticales que le permiten traducir frases que no vio antes.

Los modelos de traducción utilizan la secuencia en el idioma de origen y la traducción generada hasta el momento para asignar a cada palabra en el idioma de destino la probabilidad de que sea la siguiente palabra en la traducción.

Un detalle importante es que los modelos de aprendizaje automático trabajan con datos numéricos: reciben números como entrada y producen números como salida. En el caso de la traducción automática, el texto debe ser transformado en números para que el modelo pueda procesarlo correctamente. Esto se logra mediante el uso de *word embeddings*. Un word embedding es una representación numérica de una palabra en forma de vector, donde a cada palabra se le asigna un vector único de dimensión fija n.

Una característica fundamental de los word embeddings es que los vectores correspondientes a palabras con significados similares tienden a estar más cercanos entre sí en el espacio vectorial. De este modo, los word embeddings capturan relaciones semánticas entre palabras, proporcionando una representación significativa que los modelos pueden aprovechar.

En la Figura 2.2 se presenta del lado izquierdo un ejemplo de embeddings de dimensión 7 para las palabras cat (gato), kitten (gatito), dog (perro) y houses (casas). Del lado derecho de la Figura, se muestra la representación gráfica en dos dimensiones de los vectores: se puede observar que las palabras cat y kitten están más cercanas entre si que cat y dog, que a su vez están más cercanas entre si que cat y houses.

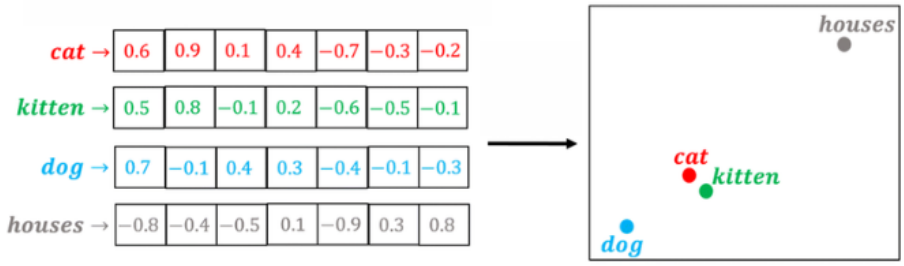


Figura 2.2: Ejemplo de Word Embedding. Imagen de Om (2023).

Los word embeddings se obtienen mediante modelos de aprendizaje automático, que se entrenan con *corpus* de textos (conjuntos de textos), utilizando usualmente una de dos estrategias principales. Una estrategia es estudiar la coocurrencia de palabras en los textos de entrenamiento: las palabras que ocurren juntas más veces serán más cercanas entre sí. Otra estrategia consiste en ocultar una palabra en una oración y entrenar al modelo para predecirla.

En el contexto de la traducción automática, una de las arquitecturas más innovadoras y eficaces es el Transformer, que ha demostrado ser especialmente efectiva en tareas de traducción automática, ya que permite aprender dependencias a largo plazo entre las palabras de una frase, superando las limitaciones de los modelos anteriores. En la siguiente sección, se explica con detalle esta arquitectura.

2.6. Arquitectura Transformer

El Transformer es una arquitectura de aprendizaje automático presentada en Vaswani et al. (2017) que fue revolucionaria para tareas de procesamiento de lenguaje natural, como la traducción automática. Su desempeño superó ampliamente a arquitecturas anteriores, y hoy en día tecnologías como Google Translate usan variantes de esta arquitectura.

A diferencia de otros modelos que generan toda la secuencia de salida de una sola vez, el Transformer genera la salida de manera secuencial, palabra por palabra. En cada paso, el modelo selecciona el siguiente *token* (o palabra) en la secuencia de salida en función de los tokens previos generados y la frase de entrada recibida.

En la Figura 2.3 se presenta un esquema de la arquitectura con sus componentes, los cuales serán explicados en detalle en esta sección. Se observan dos grandes componentes: El *Encoder* (del lado izquierdo), y el *Decoder* (del lado derecho).

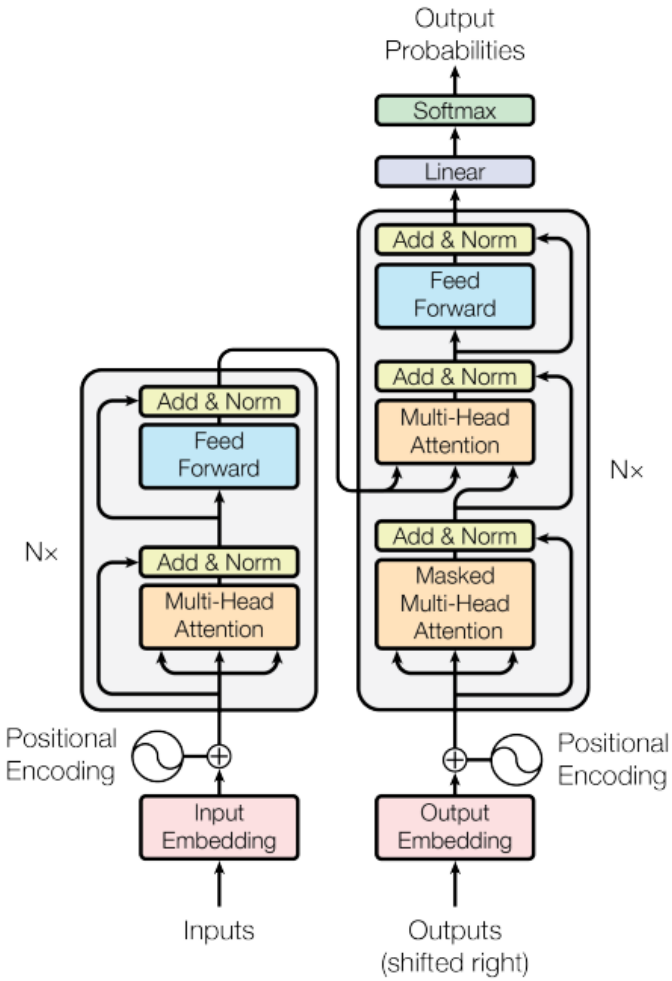


Figura 2.3: Arquitectura Transformer,
con Encoder del lado izquierdo y Decoder del lado derecho.
Imagen de Vaswani et al. (2017).

El Encoder procesa la frase original, capturando la información relevante y generando una representación interna rica de los elementos de entrada. Este bloque está diseñado para comprender las relaciones entre los diferentes componentes de la entrada y su contexto

en la frase. Antes de que la entrada llegue al Encoder, se realiza un preprocesamiento que incluye la tokenización, un proceso en el cual la frase se divide en unidades más pequeñas, como palabras o subpalabras. Luego, estos tokens se convierten en vectores numéricos mediante embeddings, para que el modelo trabaje con representaciones matemáticas.

Por otro lado, el Decoder toma la representación generada por el Encoder y la secuencia de salida generada hasta el momento, y las utiliza para producir una distribución de probabilidades para cada token, que representan la certeza del modelo sobre cuál es el token más probable para ocupar la siguiente posición en la secuencia de salida. Al igual que en el Encoder, la entrada del Decoder también pasa por un preprocesamiento, donde los tokens generados previamente se convierten nuevamente en vectores.

Para obtener la traducción completa, se van a elegir uno por uno los tokens a agregar a la secuencia de salida, basándose en la distribución de probabilidad generada por el Decoder. Una vez seleccionado el siguiente token, se ingresa nuevamente al Decoder la salida del Encoder y la secuencia de salida, ahora extendida con un token más. Así, el Decoder genera nuevamente las probabilidades teniendo en cuenta el token ya elegido. Estas probabilidades se utilizan para seleccionar el siguiente token de la secuencia, y así sucesivamente hasta generar la traducción completa. Este proceso de generar tokens de uno en uno y retroalimentar al Decoder permite que el modelo actualice las probabilidades teniendo en cuenta tanto la información de la frase original como la información brindada por la secuencia de salida generada hasta el momento.

En este trabajo, nos centramos en el uso de la arquitectura Transformer para un modelo de traducción. Para entrenar el modelo, se usa el mecanismo de aprendizaje supervisado esbozado en la Sección 2.3.1, en el cual el conjunto de entrenamiento está compuesto por pares de frases en los idiomas origen y destino. Durante el proceso de entrenamiento, se actualizan los valores de los parámetros del modelo con el objetivo de que, para cada entrada del conjunto de entrenamiento, la salida del modelo se acerque lo más posible a la frase correspondiente en el idioma destino.

Una vez entrenado el modelo, y fijados los valores de sus parámetros, lo usamos para inferencia. Esto es, para obtener la traducción de frases que el modelo no vio en el entrenamiento. En este proceso, cuando el modelo recibe una frase en el idioma de origen, esta es procesada por el Encoder, generando una representación interna. Luego, el Decoder toma esta representación y, de manera iterativa, selecciona un token y lo agrega a la secuencia de salida, repitiendo el proceso hasta obtener la traducción completa. A continuación, se explica en detalle cada etapa de un modelo Transformer, analizando cómo se utiliza para traducir una frase de un idioma de origen a uno de destino durante la inferencia.

2.6.1. Preprocesamiento de la entrada del Encoder

Al recibir una frase en el idioma de origen, lo primero que hace el Transformer es dividirla en tokens. Este proceso es realizado por un tokenizador, que divide la frase en unidades más pequeñas como palabras o subpalabras. El tokenizador utiliza un vocabulario predefinido, que es el conjunto de tokens conocidos por el modelo. Así, la frase de entrada se divide en los tokens que forman parte de este vocabulario.

Consideremos, por simplicidad, que cada palabra corresponde a un token, pero en la práctica muchas veces cada palabra se descompone en distintos tokens. Además, se agregan dos tokens especiales: '⟨SOS⟩' (Start Of Sequence) y '⟨EOS⟩' (End Of Sequence)

que se utilizan para marcar el inicio y final de la frase respectivamente. Por ejemplo, si la frase de entrada es “El pasto es verde” (y cada una de estas palabras está en el vocabulario del tokenizador), tendríamos ahora una lista de la forma [‘⟨SOS⟩’, ‘El’, ‘pasto’, ‘es’, ‘verde’, ‘⟨EOS⟩’]. El largo de la lista es el largo de la frase de entrada más dos, y la i-ésima palabra de la frase de entrada corresponde con el (i+1)-ésimo elemento de la lista.

Además, se utiliza un token de padding ‘⟨PAD⟩’ para igualar la longitud de las frases. Se define una longitud máxima de secuencia (en términos de cantidad de tokens), y si la frase de entrada es más corta que el largo máximo, se añaden tantos tokens de padding como sea necesario, después del token ‘⟨EOS⟩’, hasta alcanzar la longitud máxima.

Luego, cada token se convierte en un vector de embedding de dimensión fija, mediante un algoritmo de embedding. Esto resulta en una matriz de dimensión *largo_máximo* × *dimensión_del_embedding*, donde cada fila corresponde al embedding de un token en la frase de entrada. Esta representación vectorial inicial se denomina *input embedding*.

En la Figura 2.4 se resume el proceso de transformar el texto de entrada en el input embedding. En este ejemplo, se toma 7 como el largo máximo de secuencia, con lo cual es necesario agregar exactamente un token de padding.

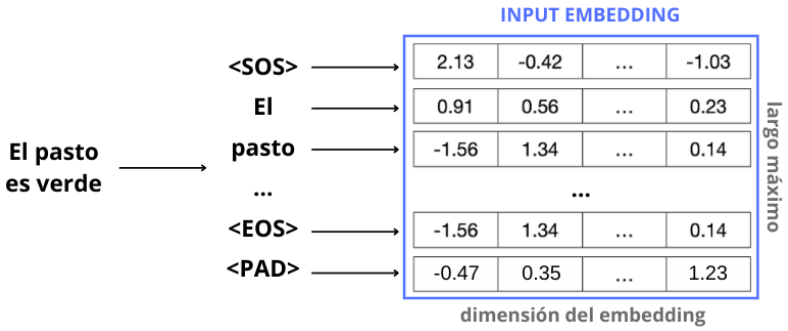


Figura 2.4: Obtención de Input Embedding

En el Encoder, los tokens se procesan en paralelo, lo que hace que se pierda información sobre su posición en la frase, algo fundamental para su significado (por ejemplo, “el gato persigue al perro” no es igual a “el perro persigue al gato”). Es necesario entonces agregar información sobre el orden de las palabras, es decir, la posición que ocupa cada palabra en la frase. Para esto, los Transformers usan un mecanismo llamado *positional encoding* (o codificación posicional), que agrega a cada token información sobre su posición en la frase. Es necesario que el positional encoding cumpla los siguientes requisitos:

1. A cada posición le corresponde un vector distinto.
2. La distancia entre los vectores de dos posiciones determinadas debe mantenerse sin importar el largo de la frase.
3. Debe funcionar con frases de cualquier longitud.
4. Debe ser determinista. El encoding para una determinada posición siempre es el mismo, independientemente del largo de la frase.

En el Transformer original, el positional encoding utiliza funciones seno y coseno para construir una matriz $P \in R^{l \times d}$ (con l la longitud máxima y d la dimensión del embedding).

Dada una posición *pos* y una dimensión *i*, el positional encoding está dado por:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d}) \tag{2.7}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d}) \tag{2.8}$$

El positional encoding se suma al input embedding obtenido anteriormente, agregando así al embedding de cada token, información sobre la posición que ocupa en la frase.

En modelos reales, con embeddings de alta dimensión y positional encodings sinusoidales, es muy poco probable que dos combinaciones distintas de palabra y posición generen el mismo vector. Cada palabra en una posición específica tiene un vector único, lo que permite al modelo reconocer tanto la identidad de la palabra como su posición, manteniendo la información contextual sin ambigüedades.

En la Figura 2.5 se presentan los valores del positional encoding, para una frase con longitud máxima de seis tokens, y dimensión del embedding de cinco.

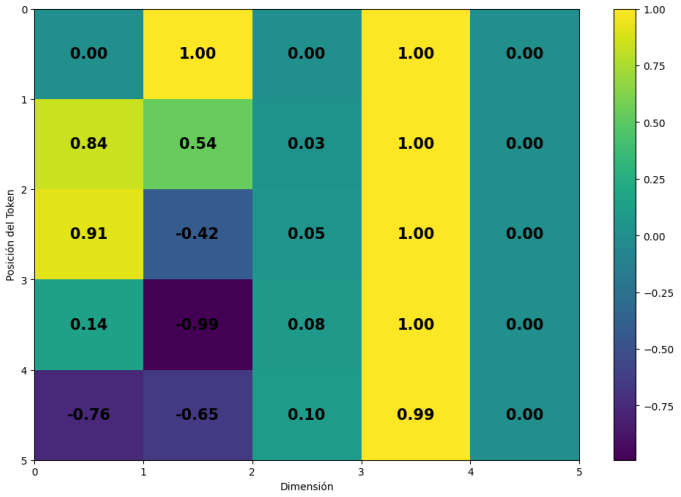


Figura 2.5: Positional Encoding con largo 6 y dimensión 5.

Para observar este comportamiento, consideremos dos frases: “El pasto es verde” y “Hay mucho pasto”, y tomemos un largo máximo de secuencia de 6 tokens. Al tokenizar estas frases obtenemos [‘⟨SOS⟩’, ‘El’, ‘pasto’, ‘es’, ‘verde’, ‘⟨EOS⟩’] y [‘⟨SOS⟩’, ‘Hay’, ‘mucho’, ‘pasto’, ‘⟨EOS⟩’, ‘⟨PAD⟩’]. En la Figura 2.6 se presenta un posible input embedding de tamaño 5 para la primera frase, y el resultado de sumarle el positional encoding presentado en la Figura 2.5 (que corresponde a embeddings de largo 5 y largo máximo de secuencia 6). En la Figura 2.7 se presenta lo mismo pero para la segunda frase.

	Input Embedding				Input Embedding + Embedding Posicional			
<SOS>	2.13	-0.42	...	-1.03	2.13	0.58	...	-1.03
El	0.91	0.56	...	0.23	...			
pasto	-1.56	1.34	...	0.14	-0.65	0.92	...	0.14
...	...				...			

Figura 2.6: Embedding para ‘El pasto es verde’

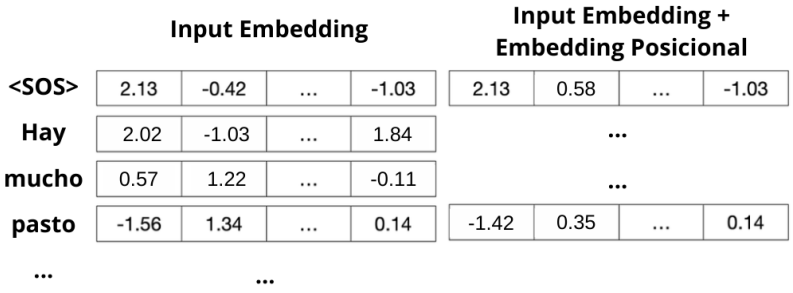


Figura 2.7: Embedding para ‘Hay mucho pasto’

Notar que en ambas frases, el embedding final para ‘⟨SOS⟩’ es el mismo, pues es el mismo token en la misma posición en ambas frases. Sin embargo, el embedding final para ‘pasto’ es diferente en cada frase. Aunque el input embedding para este token es el mismo, el positional encoding varía según su posición en la secuencia, obteniendo embeddings finales distintos. En conclusión, al sumar el positional encoding al input embedding, se obtiene información tanto sobre el token como sobre su posición, lo que facilita al modelo captar relaciones contextuales clave en las secuencias de entrada.

Una vez obtenido el embedding final de la frase en el idioma de origen, este será pasado como entrada al Encoder. A continuación, se detallan los componentes del Encoder.

2.6.2. Encoder

El Encoder está compuesto por una cantidad fija de bloques de Encoder, todos iguales. Cada bloque de Encoder realiza las siguientes etapas:

1. Atención con múltiples cabezales.
2. Conexión residual y normalización.
3. Red neuronal feed forward fully connected.
4. Conexión residual y normalización.

En la Figura 2.8 se esquematiza la arquitectura de un bloque de Encoder.

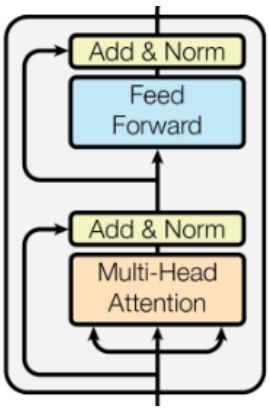


Figura 2.8: Bloque de Encoder. Imagen de Vaswani et al. (2017).

A continuación, se explica el funcionamiento de cada uno de los componentes enumerados.

Atención con múltiples cabezales

El mecanismo de atención permite al modelo identificar las palabras más relevantes al analizar una palabra en particular dentro de una secuencia. Es decir, la atención ayuda al modelo a "fijarse" en las palabras que aportan más significado en el contexto actual. Por ejemplo, en la frase "Hoy fui al banco a sacar plata", al analizar la palabra 'banco', es crucial considerar la palabra 'plata' para poder inferir el significado en este contexto.

Para poder entender el mecanismo de atención de múltiples cabezales, primero nos concentramos en la atención simple (*single-head attention*), que utiliza un único cabezal. El primer paso en la atención simple es generar tres vectores a partir del vector de embeddings de cada token en la frase: el vector de *consulta* (*query*), el vector de *clave* (*key*) y el vector de *valor* (*value*). Estos vectores se obtienen multiplicando el vector de embedding de cada token por tres matrices de pesos distintas: la matriz de consulta, la matriz de clave y la matriz de valor. Cada una de estas matrices es un parámetro del modelo, cuyos valores se aprenden en el proceso de entrenamiento. La entrada al bloque es una matriz de tamaño $l \times d$, con l el largo máximo de la secuencia y d la dimensión del embedding. Las matrices de pesos serán de tamaño $d \times d$ cada una, así cada vector (ya sea de consulta, clave o valor) será de tamaño d. Denotamos d_q, d_k y d_v a la dimensión de los vectores de consulta, clave y valor respectivamente.

En la Figura 2.9 se muestra cómo obtener los vectores de consulta, clave y valor para una entrada X. Cada fila de la matriz X corresponde con el embedding de entrada para una palabra de la frase, y cada fila en Q, K, y V corresponde al vector de consulta, clave y valor respectivamente para cada palabra.

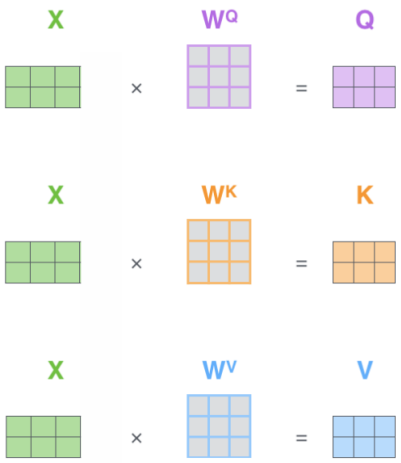


Figura 2.9: Obtención de vectores de consulta, clave y valor.
Imagen de Alammar (2018).

Consideremos un ejemplo para entender de forma más intuitiva qué representa cada uno de estos vectores. Para buscar un video en Youtube, se introduce la consulta en la barra de búsqueda. Youtube va a comparar esta consulta con un conjunto de claves, como

puede ser el nombre de los videos o canales, etiquetas, etc. Luego, nos devuelve los videos tales que sus claves coinciden con nuestra consulta, el video en sí es el valor.

Una vez obtenidos los tres vectores (consulta, clave y valor) para cada palabra en la secuencia, el siguiente paso es calcular la atención de manera paralela para todas las palabras. Nos centraremos en el cálculo de la atención para una palabra específica, a la que llamaremos *palabra actual*.

Primero se calcula el *puntaje de similitud* entre la palabra actual y cada palabra en la frase, incluida ella misma. Para esto, tomamos el producto punto entre el vector de consulta de la palabra actual y el vector clave de cada una de las palabras de la frase. Esto produce un conjunto de puntajes que representan la "afinidad" de la palabra actual con cada palabra de la frase.

A continuación, el puntaje de similitud se divide por $\sqrt{d_k}$ (8 en el paper original), lo cual ayuda a la estabilización. Luego se aplica una función *softmax* sobre los puntajes para convertirlos en una distribución de probabilidades. Aplicar la función softmax asegura que los puntajes se normalicen entre 0 y 1, y que su suma sea igual a 1. La probabilidad resultante para cada palabra indica qué tan relevante es para la palabra actual.

Finalmente, se multiplica el vector de valor de cada palabra por su respectivo puntaje de atención (el valor obtenido tras aplicar softmax). Después, sumamos todos los vectores resultantes de estas multiplicaciones ponderadas para obtener el vector de salida de atención para la palabra actual. Este vector de salida es una combinación ponderada de los vectores de valor de todas las palabras en la secuencia, donde el peso de cada palabra depende de su relevancia con respecto a la palabra actual.

En la Figura 2.10 se resume lo explicado, usando como ejemplo el cálculo de atención para la palabra 'Thinking' en la frase de entrada "Thinking Machines".

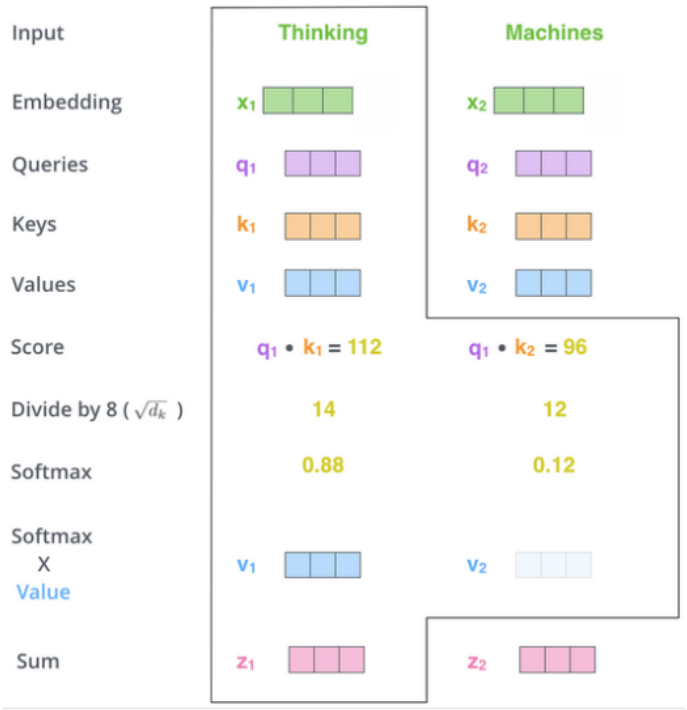


Figura 2.10: Cálculo de atención sobre una palabra. Imagen de Alammar (2018)

Este proceso se repite para cada palabra en la frase, generando un conjunto de embed-

dings contextualizados que el modelo utilizará en las siguientes capas.

La atención con múltiples cabezales simplemente tiene múltiples conjuntos de matrices de consulta, valor y clave; y realiza atención simple con cada uno de ellos por separado. Cada cabezal aprende a enfocarse en distintos aspectos del contexto, capturando diferentes patrones de relación entre palabras.

Una diferencia fundamental con respecto a la atención simple es la reducción del tamaño de las proyecciones. En lugar de realizar la atención en el espacio original de los embeddings, de dimensión (d), cada cabezal trabaja en un subespacio más pequeño de dimensión $d_k = d/h$ donde h es el número de cabezales. Entonces, cada cabezal proyecta las entradas en subespacios de tamaño reducido mediante transformaciones lineales.

Una vez calculadas las salidas de todas las cabezales, estas se concatenan para formar un vector de dimensión d, que luego pasa por otra transformación lineal utilizando una matriz de pesos W_o. Esta última operación combina las contribuciones de todas las cabezales, generando una representación final que captura múltiples perspectivas de las relaciones entre las palabras de la secuencia.

Una vez obtenidos los vectores finales de atención para cada palabra, estos pasan por una conexión residual y normalización, proceso que se describe a continuación.

Conexión residual y normalización

Esta componente aplica dos operaciones para mejorar la estabilidad y eficiencia del modelo. Primero, se realiza una conexión residual, que suma la salida del módulo anterior a la entrada original del módulo. Esto ayuda a preservar características relevantes de la información inicial Luego, la suma resultante pasa por un proceso de normalización por capas (*layer normalization*), que ajusta las salidas restando su media y dividiendo por su desviación estándar a lo largo de la dimensión de características. Luego, la salida se reescala mediante parámetros aprendibles. Esto mejora la estabilidad y facilita la convergencia del modelo durante el entrenamiento (Ba et al., 2016).

Este proceso prepara las representaciones para ser procesadas por la siguiente capa: la red neuronal feed forward fully connected, que se describe a continuación.

Red neuronal feed forward fully connected

La salida de la normalización será la entrada para una red neuronal feed forward fully connected. Esta red tiene una única capa oculta, con función de activación $ReLU$; y las capas de entrada y salida son lineales, no aplican ninguna función de activación.

El objetivo principal de esta red es aplicar una transformación no lineal a las representaciones generadas por la atención, permitiendo al modelo aprender relaciones más complejas y generar representaciones enriquecidas antes de pasar al siguiente bloque.

Al finalizar este paso, la salida pasa nuevamente por una capa de conexión residual y normalización y con esto se concluye un bloque del Encoder. Si hay más bloques, la salida de esta segunda normalización será utilizada como entrada para el siguiente bloque. En caso de ser el último bloque, la salida será la salida final del Encoder.

Una vez que la frase en el idioma de origen ha sido procesada por el Encoder, el siguiente paso en el modelo de Transformer es el Decoder, que utiliza tanto la representación generada por el Encoder como la información de la traducción parcial para generar la secuencia en el idioma de destino.

2.6.3. Preprocesamiento de la entrada del Decoder

El Decoder recibe dos entradas principales: la salida del Encoder y la traducción parcial generada hasta el momento. La salida del Encoder es la representación generada a partir de la frase en el idioma de origen, que captura las relaciones y el contexto entre las palabras de la secuencia original. Por otro lado, la traducción parcial contiene información sobre las palabras ya generadas en el idioma de destino durante pasos anteriores. En el primer paso del proceso de traducción, cuando aún no se ha generado ninguna palabra en el idioma de destino, la "traducción hasta el momento" que se usará como entrada del Decoder será simplemente el token '⟨SOS⟩'.

Al igual que en el Encoder, la secuencia generada hasta el momento pasa por un proceso de tokenización. Se añade el token especial '⟨SOS⟩', pero no el token '⟨EOS⟩', ya que se quiere que el modelo aprenda a identificar cuándo la frase debe terminar, es decir, cuándo debe agregar '⟨EOS⟩'.

Es importante señalar que el vocabulario del tokenizador del Encoder y el del Decoder son diferentes. Tendremos dos tokenizadores separados, cada uno con su propio vocabulario. El tokenizador del Encoder (o tokenizador de entrada) está diseñado para manejar frases en el idioma de origen, con lo cual su vocabulario está formado por las palabras o subpalabras de dicho idioma. En cambio, el tokenizador del Decoder (o tokenizador de salida) trabaja con secuencias en el idioma de destino, por lo que su conjunto de tokens está formado por palabras y subpalabras de dicho idioma.

Una vez obtenidos los tokens, se calcula el embedding de cada uno, y obtenemos una matriz donde cada fila es el embedding de un token en particular, esta matriz se llama output embedding. A continuación, al igual que en el preprocesamiento de la entrada del Encoder, aplicamos un encoding posicional para incorporar la información sobre la posición de los tokens en la secuencia. La suma de estos dos vectores (embeddings y encoding posicional) constituye la entrada que se alimenta al Decoder.

2.6.4. Decoder

El Decoder por su parte tiene una cantidad fija de bloques de Decoder, todos iguales. Cada bloque de Decoder tiene las siguientes etapas:

1. Atención con múltiples cabezales y máscara.
2. Conexión residual y normalización.
3. Atención con multiples cabezales cruzada.
4. Conexión residual y normalización.
5. Red neuronal feed forward fully connected.
6. Conexión residual y normalización.

En la Figura 2.11 se esquematiza la arquitectura de un bloque de Decoder.

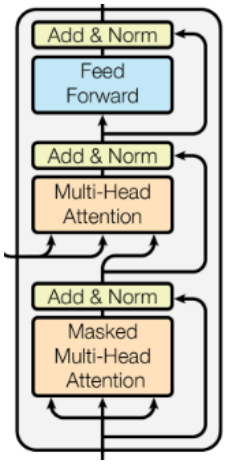


Figura 2.11: Bloque de Decoder. Imagen de Vaswani et al. (2017).

A continuación, se explica cada uno de los componentes enumerados.

Atención con múltiples cabezales y máscara

En el Decoder, se aplica una atención con múltiples cabezales, como se explicó para el Encoder. En este caso, se aplica sobre la secuencia de salida generada hasta el momento, es decir, la traducción parcial. Además, se introduce una *máscara*, cuyo objetivo y funcionamiento se detallan a continuación.

Durante la etapa de inferencia, cuando el modelo realiza la traducción final, las palabras en el idioma destino se generan una a una de forma secuencial. Es decir, en cada paso, el modelo predice la siguiente palabra basándose en las palabras ya generadas (la traducción parcial) y la información codificada de la frase de entrada (proveniente del Encoder). Una vez generada una palabra, esta se agrega a la secuencia parcial y el proceso se repite hasta que se produce la frase completa.

Sin embargo, durante el entrenamiento, ya tenemos los pares completos de frases en el idioma origen y en el idioma destino. Esto, en principio, permitiría calcular en paralelo todas las palabras en la salida (pues para cada palabra ya están todas las anteriores generadas desde el comienzo), en lugar de generarlas una por una como se hace en inferencia.

A pesar de esta posibilidad de paralelización, se introduce una máscara para asegurar que el modelo respete el flujo secuencial de generación de palabras. Esta máscara tiene como objetivo bloquear el acceso a posiciones futuras dentro de la secuencia de salida durante el cálculo de la atención. En concreto, cuando el modelo calcula la atención para una palabra en una posición específica, la máscara garantiza que sólo pueda "ver" y utilizar las palabras de posiciones anteriores. Las palabras en posiciones futuras son ocultadas asignando pesos de atención nulos (o menos infinito), impidiendo que el modelo utilice información que no estaría disponible durante la inferencia real.

Sin esta máscara, el mecanismo de atención podría "mirar al futuro" y utilizar información de palabras aún no generadas, lo que invalidaría el proceso secuencial de generación y rompería el flujo natural de la inferencia. La máscara garantiza que el modelo prediga cada palabra solo con la información disponible hasta ese momento, simulando cómo funcionará durante la inferencia.

La salida de esta etapa pasa por una conexión residual, que se explica a continuación.

Conexión residual y normalización

La etapa de conexión residual y normalización se realiza igual que en el Encoder. La conexión residual agrega la entrada de la capa de atención a su salida, y luego la normalización ajusta las salidas.

Atención con multiples cabezales cruzada

En la atención con múltiples cabezales explicada anteriormente, los vectores de clave, valor y consulta se generaban para cada uno de los tokens de la secuencia de entrada; ya sea la secuencia en el idioma origen para el Encoder o la traducción parcial en el idioma destino en la atención con máscara para el Decoder. Esto permite encontrar relaciones entre los tokens de una misma frase consigo misma.

Sin embargo, al momento de realizar una traducción, no sólo queremos capturar relaciones entre las palabras del idioma destino, sino también analizar cómo estas se relacionan con las palabras en el idioma origen. Por ejemplo, si estamos traduciendo la frase "El pasto es verde" al inglés, para generar la palabra 'green' es necesario que el modelo preste atención a la palabra 'verde', es la que aporta el significado relevante.

La atención cruzada (*cross-attention*) busca justamente que se preste atención a las palabras de la frase en el idioma origen al momento de determinar el valor de las palabras de la frase en el idioma destino. Para lograr esto, los vectores de consulta se generan a partir de la traducción parcial (los tokens generados hasta el momento en el idioma destino), mientras que los vectores de clave y valor se calculan utilizando los tokens de la representación generada por el Encoder para la frase en el idioma origen.

Es importante notar que para la atención cruzada no es necesario utilizar una máscara, ya que en esta instancia los vectores de clave y valor se obtienen a partir de la salida del Encoder, que ya fue generada en su totalidad. Así, cuando estamos posicionados sobre una palabra, su vector de consulta se "combina" con vectores generados a partir del Encoder.

En la Figura 2.12 se presenta la parte del diagrama general de la arquitectura en la que se muestra que la atención cruzada en el Decoder recibe las matrices de clave y valor se obtienen de la salida del Encoder, mientras que la de Consulta proviene de la etapa anterior del Decoder.

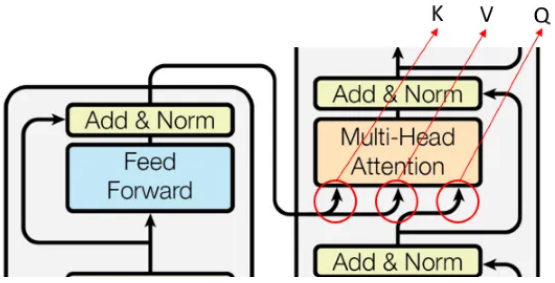


Figura 2.12: Cálculo de atención sobre matriz de entrada.
Imagen de Vaswani et al. (2017).

Luego de esta capa de atención, nuevamente se realiza una conexión residual seguida de normalización. La salida de esta operación se utiliza luego como entrada para una red neuronal feed forward fully connected, que se detalla a continuación.

Red neuronal feed forward fully connected

Al igual que en el Encoder, se aplica una red neuronal cuya función es transformar las representaciones obtenidas tras la etapa de atención en representaciones más ricas y no lineales, lo que permite al modelo capturar interacciones complejas entre las características de cada token.

Al finalizar este paso, la salida pasa nuevamente por una capa de conexión residual y normalización y con esto se concluye un bloque del Decoder. Si hay más bloques, la salida de esta tercera normalización será utilizada como entrada para el siguiente bloque. En caso de ser el último bloque, la salida será la salida final del Decoder.

A continuación, se explica cómo se transforma la salida del Decoder en una asignación de probabilidades a cada token del vocabulario del tokenizador de salida, y cómo se utilizan estas probabilidades para construir la traducción final.

2.6.5. Procesamiento final y generación de la traducción

Al finalizar el proceso de decodificación, la salida del último bloque del Decoder pasa por una capa lineal, que transforma la salida en un vector de dimensión $|\mathscr{V}_{\text{tgt}}|$, donde $|\mathscr{V}_{\text{tgt}}|$ es el tamaño del vocabulario del tokenizador del Decoder (es decir, el número de tokens posibles en el idioma de destino).

Este vector contiene un puntaje (*logit*) para cada token en el vocabulario de salida. A mayor puntaje, mayor es la confianza del modelo de que ese token debería ser el siguiente en la traducción. A continuación, se aplica una función softmax a estos puntajes, transformándolos en probabilidades que suman 1. Así, obtenemos un vector de probabilidades, donde cada valor representa la probabilidad de que un token sea el siguiente en la secuencia de traducción.

Vemos entonces que dada una frase en el idioma origen, y una traducción parcial (que podría ser vacía), el Transformer simplemente devuelve la probabilidad para cada palabra del idioma destino de ser la próxima palabra de la traducción.

Entonces, para traducir una frase de un idioma origen a un idioma destino utilizando un modelo Transformer, se pasa por las siguientes etapas:

Procesamiento de la frase en el idioma origen (Encoder): La primera etapa es pasar la frase en el idioma de origen a través de los bloques Encoder. La salida es una representación de alto nivel de la entrada, que contiene información contextual de la frase original.

Inicialización de la frase destino: Al comienzo del proceso de traducción, cuando todavía no se generó ningún token, la frase en el idioma de destino tendrá solamente el token '$\langle SOS \rangle$' de inicio de secuencia. Por lo tanto, la entrada al Decoder en el primer paso es la salida del Encoder (obtenida en el paso anterior) y la frase destino ['$\langle SOS \rangle$'].

Generación de probabilidades de tokens del idioma destino (Decoder): Con la salida del Encoder y la frase destino vacía como entrada, el Decoder procesa esta información y genera una distribución de probabilidad sobre todos los tokens posibles en el vocabulario del idioma destino. Cada token del vocabulario recibe una probabilidad que indica cuán probable es que sea el siguiente token de la traducción.

Selección del siguiente token: Luego, se selecciona el token con la mayor probabilidad. Este token se agrega como el primer token de la traducción en el idioma de destino.

Repetición del proceso: Ahora, con un token agregado a la frase destino, la nueva entrada para el Decoder consiste en la salida del Encoder (que no cambia) y la frase

destino actualizada. El modelo repite el proceso de generar probabilidades para todos los tokens del idioma destino, eligiendo el token con mayor probabilidad y agregándolo a la frase destino. Este proceso se repite hasta que el modelo genera el token de fin de frase, lo que indica que la traducción está completa, o hasta que se alcanza un largo máximo de secuencia permitido.

El proceso explicado corresponde a lo que se llama una traducción *greedy*, en la que en cada paso se elige el token con la mayor probabilidad. Esto puede hacer que el modelo se quede atascado en traducciones subóptimas, especialmente en casos donde una palabra en el idioma de destino tiene múltiples interpretaciones, dependiendo de las palabras que la rodean.

En contraste, Beam Search es una alternativa más compleja pero generalmente más precisa. En lugar de elegir solo el token con mayor probabilidad en cada paso, Beam Search mantiene múltiples opciones de traducción. Esto permite que el modelo explore diferentes caminos de traducción simultáneamente y luego seleccione la mejor opción.

Capítulo 3

Descripción del Problema y Solución Propuesta

En este capítulo se describe en detalle el problema central abordado en este trabajo, junto con el enfoque desarrollado para resolverlo. El problema principal se relaciona con la generación eficiente de una representación en STRIPS a partir de problemas en PDDL.

Para esto, se introduce el concepto de *grounding*, que consiste en instanciar predicados y esquemas de acciones de un problema en PDDL para obtener hechos y operadores que pueden ser utilizados en una representación STRIPS del problema.

Se presentan tres estrategias principales de grounding: el *grounding total*, que considera todas las combinaciones posibles de objetos para instanciar predicados y esquemas de acciones; el *grounding clásico*, que limita la instanciación a hechos y acciones alcanzables de forma relajada desde el estado inicial; y el *grounding parcial con criterio heurístico*, que es el foco central de este trabajo.

Este último enfoque busca reducir la cantidad de hechos y acciones instanciados descartando aquellos que probablemente no sean relevantes para encontrar un plan del problema. Para esto, hace uso de una función llamada $popBestOp$, que, dado un problema, asigna a cada operador una probabilidad de ser parte de un plan que resuelva el problema. Luego, se priorizan aquellas instancias con mayor probabilidad de contribuir a la solución.

En este trabajo, se propone para la definición de la función $popBestOp$ el uso de un un modelo de traducción automática basado en la arquitectura Transformer, para traducir hechos relajados de un problema, en un plan del mismo. En este capítulo, se explica la idea de este modelo, y cómo se procesa la salida del mismo para calcular las probabilidades que asigna la función $popBestOp$. Además, se presenta la métrica PUO, utilizada para evaluar el desempeño de la función. El capítulo concluye con un ejemplo de uso de la función $popBestOp$, donde se muestra concretamente el cálculo de probabilidades a partir de la salida del modelo.

3.1. Grounding

En el capítulo anterior se presentaron dos lenguajes usados para representar problemas de planning: STRIPS y PDDL. Se destacó que PDDL permite representar los problemas de manera más compacta que STRIPS, ya que trabaja con esquemas que contienen variables, en lugar de enumerar explícitamente todos los hechos y operadores.

Por ejemplo, en el dominio de la aerolínea, en STRIPS sería necesario especificar de manera explícita hechos como $\mathit{TieneCombustible}(\mathit{Avión1})$ y $\mathit{TieneCombustible}(\mathit{Avión2})$, escribiéndolos individualmente en el problema. En cambio, en PDDL, se define el predicado $\mathit{TieneCombustible}(\mathit{avión})$, donde *avión* es una variable. Al instanciar esta variable con objetos concretos, como *Avión1* o *Avión2*, se obtienen los hechos.

Esto no sólo simplifica la escritura de problemas, sino que también hace que PDDL sea más escalable: en escenarios grandes con numerosos objetos, enumerar manualmente todos los predicados y operadores en STRIPS es inviable, mientras que PDDL permite una descripción más simple y elegante.

Si bien en los últimos tiempos se han explorado formas de encontrar planes directamente con las representaciones en PDDL (Corrêa et al., 2020; Lauer et al., 2021; Ridder and Fox, 2014), las heurísticas de búsqueda más eficientes se basan en las representaciones STRIPS. Es por esto que la mayoría de los planners actuales, como Fast Forward (Hoffmann and Nebel, 2011) y LAMA (Richter et al., 2011), se basan en representaciones instanciadas de los problemas.

Entonces, si tenemos un problema en PDDL, es necesario pasarlo a una representación en STRIPS para que pueda ser procesado. Este proceso, conocido como grounding, consiste en instanciar (es decir, reemplazar las variables por objetos concretos) los predicados y esquemas de acción, generando así los hechos y operadores necesarios para la representación STRIPS. El grounding es una etapa previa fundamental al proceso de búsqueda de la solución (*searching*) realizado por el planner.

Formalmente, el grounding es el proceso de transformar un problema representado en PDDL ($\Pi = \langle P, A, \Sigma^C, \Sigma^O, I, G\rangle$) a su equivalente en STRIPS ($\Pi = \langle F, O, I, G\rangle$), instanciando los predicados y esquemas de acciones con los objetos del dominio (Σ^C). Hay diversas estrategias de grounding, que se distinguen por la cantidad de hechos y acciones que instancian. A continuación se presentan tres estrategias.

3.1.1. Grounding total

En el grounding total, se instancian todos los predicados de P y todos los esquemas de acción de A utilizando todas las combinaciones posibles de objetos de Σ^C. De esta manera, F será el conjunto de hechos que se obtiene al instanciar cada predicado $p[X] \in P$ con todas las combinaciones posibles de objetos de Σ^C. Por otro lado, O será el conjunto de operadores que se obtiene al instanciar cada esquema de acción $a[X] \in A$ con todas las combinaciones posibles de objetos de Σ^C. Formalmente, esto se define como:

$$F = P^{\Sigma^C} = \{p(o_1, o_2, \dots) : p \in P, o_i \in \Sigma^C\}.$$

$$O = A^{\Sigma^C} = \{a(o_1, o_2, \dots) : a \in A, o_i \in \Sigma^C\}.$$

Es decir, denotamos P^{Σ^C} al resultado de instanciar todos los predicados de P con todas las combinaciones posibles de objetos en Σ^C, y de forma análoga denotamos A^{Σ^C} al resultado de instanciar todos los esquemas de acciones de A con todas las combinaciones posibles de objetos en Σ^C. Notar entonces que P^{Σ^C} es un conjunto de hechos, y A^{Σ^C} es un conjunto de operadores.

Esta estrategia permite obtener una instanciación completa del problema, instanciando todas las acciones y hechos posibles. Sin embargo, este proceso genera una gran cantidad

de acciones y operadores, muchos de los cuales no serán relevantes para el plan final.

Además, este proceso puede generar operadores y hechos mal tipados, es decir se instancian variables con objetos de un tipo incorrecto. En el dominio de la aerolínea, se obtiene por ejemplo $CargarCombustible(Ezeiza)$ o $Volar(IngAT, Avión1, Avión2)$. Aunque PDDL permite restringir los tipos de cada variable, incluso respetando estas restricciones pueden generarse operadores irrelevantes, como podría ser $Volar(Avión1, IngAT, IngAT)$, que es un operador bien tipado pero que no tiene sentido en este dominio.

El grounding total puede generar una gran cantidad de operadores y hechos instanciados, lo que puede llevar a un consumo excesivo de memoria. En algunos casos puede suceder que el problema no llegue a ser procesado por el planner ya que no hay espacio suficiente para almacernar la representación. Es por esto que surge la estrategia del grounding clásico, que hace una instanciación más "inteligente", descartando la instanciación de predicados y operadores que no serán necesarios en el plan del problema. En la siguiente sección se explica esta estrategia.

3.1.2. Grounding clásico

El grounding clásico es una estrategia que busca reducir la cantidad de hechos y operadores instanciados, mientras asegura que, si existe un plan que resuelva el problema, los hechos y operadores necesarios sean instanciados.

Sea $\Pi = \langle P, A, \Sigma^C, \Sigma^O, I, G\rangle$ un problema de planning, definimos la *versión relajada* del problema como el problema $\Pi^+ = \langle P, A', \Sigma^C, \Sigma^O, I, G\rangle$ donde A' es el resultado de tomar $del(a) = \emptyset$ para cada $a \in A$. Esto es,

$$A' = \{\langle pre(a), add(a), \emptyset\rangle : a \in A\}.$$

Es decir, en el problema relajado, aplicar una acción no elimina hechos del estado, solamente los agrega. Sea Π un problema, Π^+ su versión relajada, y π^+ un plan de Π^+; decimos que π^+ es un plan relajado de Π.

Sea $\Pi = \langle P, A, \Sigma^C, \Sigma^O, I, G\rangle$ un problema de planning, diremos que un hecho $p \in P^{\Sigma^C}$ es *alcanzable* desde el estado inicial I si hay una secuencia de acciones que pasen del estado inicial a un estado que contiene al hecho. Esto es, si existen $\bar{a} \in A^{\Sigma^C *}$ y $s \subseteq P^{\Sigma^C}$ tales que $I \xrightarrow{\bar{a}} s$ y $p \in s$. Además, diremos que un hecho $p \in P^{\Sigma^C}$ es *alcanzable en forma relajada* desde el estado inicial I, si p es alcanzable desde el estado inicial I en la versión relajada de Π, es decir, en $\Pi^+ = \langle P, A', \Sigma^C, \Sigma^O, I, G\rangle$.

De manera similar, diremos que una acción $a \in A^{\Sigma^C}$ es alcanzable desde el estado inicial I si hay una secuencia de acciones que pasen del estado inicial a un estado que contenga todas las precondiciones de la acción (un estado en el que se pueda aplicar la acción). Esto es, si existen $\bar{a} \in A^{\Sigma^C *}$ y s tales que $I \xrightarrow{\bar{a}} s$ y $pre(a) \subseteq s$. Además, diremos que una acción $a \in A^{\Sigma^C}$ es alcanzable en forma relajada desde el estado inicial I si a es alcanzable desde el estado inicial I en la versión relajada de Π, es decir, en $\Pi^+ = \langle P, A', \Sigma^C, \Sigma^O, I, G\rangle$.

Es importante notar que si un hecho no es alcanzable en forma relajada en un problema, tampoco será alcanzable en ningún plan normal de un problema. Así también, las acciones que no son alcanzables en forma relajada, tampoco lo serán en ningún plan normal. Es decir, sea un problema Π, si el problema tiene un plan que lo resuelva, todas las

acciones que aparecen en el plan son alcanzables relajadamente desde el estado inicial. Con lo cual, si el problema tiene un plan que lo resuelva, con instanciar todas las acciones alcanzables relajadamente desde el estado inicial, nos aseguramos de que las acciones que aparecen en el plan fueron instanciadas.

A la vez, sea π un plan de Π, π es un plan de Π^+. Sin embargo, no todo plan de Π^+ es un plan de Π. Más aún, es posible que Π^+ tenga un plan que lo resuelva y Π no.

El grounding clásico entonces, consiste en instanciar todos los hechos y operadores alcanzables en forma relajada desde el estado inicial del problema. Esto garantiza que, si existe un plan para resolver el problema, todos los hechos y operadores necesarios estarán disponibles, mientras reduce significativamente la cantidad de elementos instanciados.

A continuación se presenta un algoritmo que, dado un problema de planning en STRIPS, devuelve una versión en PDDL del problema donde los operadores y hechos son aquellos que son alcanzables de forma relajada desde el estado inicial.

Algorithm 1

Input: Problema PDDL $(P, A, \Sigma^C, \Sigma^O, I, G)$
Output: Problema STRIPS (F, O, I, G)

```
1:  q ← Queue(I)
2:  F ← ∅
3:  O ← ∅
4:  while ¬q.empty() do
5:      e ← q.pop()
6:      if e.isFact() then
7:          F ← F ∪ {e}
8:          for a ∈ Inst(A, Obj(F), Σ^C) ∧ pre(a) ⊆ F do
9:              if a ∉ q then
10:                 q.insert(a)
11:             end if
12:         end for
13:     else
14:         O ← O ∪ {e}
15:         for f ∈ add(e) do
16:             if f ∉ q then
17:                 q.insert(f)
18:             end if
19:         end for
20:     end if
21: end while
22: return (F, O, I, G)
```

El algoritmo utiliza una cola q que contiene tanto operadores como hechos, inicializada con los hechos del estado inicial I. Durante su ejecución, los elementos de la cola se procesan y se asignan al conjunto de hechos F o al conjunto de operadores O del problema STRIPS, según corresponda.

Este algoritmo utiliza dos funciones auxiliares, Obj y Inst. En primer lugar, la función Obj toma como entrada un conjunto de hechos H y devuelve todos los objetos presentes

en los elementos de H. Por ejemplo:

$$\text{Obj}(\{Aterrizado(Avión1, Ezeiza)\}) = \{Avión1, Ezeiza\}.$$

Por otro lado, la función Inst toma un conjunto de esquemas de acciones A y dos conjuntos de objetos P y Q. Devuelve los operadores instanciados a partir de A tales que las variables que aparecen en sus precondiciones se instancian con objetos de P, y las variables que aparecen sólo en los efectos se instancian con los objetos de Q. Por ejemplo, para el esquema de acción:

$$Volar(avión, aero, aero') = \langle \text{pre} : \{Aterrizado(avión, aero), TieneCombustible(avión)\}, \ldots \rangle,$$

sólo *avión* y *aero* aparecen en las precondiciones, y *aero'* aparece en los efectos. Asi:

$$\begin{aligned}\text{Inst}(\{Volar(avión, aero, aero')\}, \{Avión1, IngAT\}, \{Ezeiza, Aeroparque\}) = \\ \{Volar(Avión1, IngAT, Ezeiza), Volar(IngAT, Avión1, Ezeiza), \\ Volar(Avión1, IngAT, Aeroparque), Volar(IngAT, Avión1, Aeroparque)\}.\end{aligned}$$

Durante la ejecución, el algoritmo extrae elementos de la cola y los procesa para agregarlos al problema STRIPS, asignándolos al conjunto de hechos F o al conjunto de operadores O, según corresponda.

- Si el elemento extraído es un hecho, este se agrega a F. Luego, se identifican todos los operadores aplicables en el estado formado por los hechos de F. Para esto, se utiliza $\text{Inst}(A, Obj(F), \Sigma^C)$ para instanciar los esquemas de acciones A de manera que los objetos en las precondiciones estén presentes en F. Luego se agregan a la cola aquellos operadores tales que sus precondiciones estén en F.

- Si el elemento extraído es un operador, este se agrega a O, y todas sus postcondiciones que no estén en F se agregan a la cola.

El proceso se repite hasta que la cola esté vacía, momento en el cual el algoritmo retorna el problema STRIPS resultante (F, O, I, G).

Notar que si bien $\text{Inst}(A, \text{obj}(F))$ está haciendo grounding total, no es sobre el conjunto de todos los objetos del problema, sino que para las variables que aparecen en las precondiciones de los esquemas, se restringe a los objetos que aparecen en F.

En conclusión, este algoritmo genera una versión en STRIPS del problema PDDL proporcionado como entrada, incluyendo únicamente los operadores y hechos que son alcanzables de forma relajada desde el estado inicial. Dado que todos los hechos y acciones alcanzables en la versión original del problema también lo son en su versión relajada, el algoritmo asegura que si existe un plan que lo resuelva, estará conformado por acciones que fueron instanciadas.

Si bien esta estrategia reduce la cantidad de hechos y operadores instanciados al considerar sólo aquellos alcanzables de forma relajada, no todos los elementos generados son necesarios para encontrar un plan que resuelva el problema. Por ejemplo, en el caso de la aerolínea, si el único objetivo del problema es $TieneCombustible(Avión1)$ y el estado inicial incluye los hechos $Aterrizado(Avión1, Ezeiza)$, $Aterrizado(Avión2, Ezeiza)$, y $TieneCombustible(Avión2)$, un plan que resuelve el problema está dado únicamente por

la acción *CargarCombustible*(*Avión1*). Pero, el operador *Volar*(*Avión2*, *Ezeiza*, *IngAT*), aunque irrelevante para el plan, sería instanciado porque es alcanzable de forma relajada desde el estado inicial. Esto muestra que, en ciertos casos, la cantidad de operadores instanciados puede seguir siendo alta, manteniendo los problemas de almacenamiento.

Para abordar esta limitación, se presenta a continuación una estrategia alternativa de grounding que utiliza un criterio heurístico, con el objetivo de reducir aún más la cantidad de hechos y operadores instanciados.

3.1.3. Grounding parcial

A continuación, se presenta un algoritmo que pasa de un problema en PDDL a un problema en STRIPS, donde los operadores y hechos instanciados se seleccionaron haciendo uso de un criterio heurístico. Este algoritmo fue presentado en Areces et al. (2023).

Algorithm 2

Input: Problema PDDL $(P, A, \Sigma^C, \Sigma^O, I, G)$
Output: Problema STRIPS (F, O, I, G)

$q \leftarrow \text{Queue}(I)$
$F \leftarrow \emptyset$
$O \leftarrow \emptyset$
while $\neg q.\text{empty}()$ **and** $\neg$ Stop() **do**
  if $q.\text{containsFact}()$ **then**
    $f \leftarrow q.\text{pop}()$
    $F \leftarrow F \cup \{f\}$
    for $a \in \text{Inst}(A, \text{obj}(F), \Sigma^C) \wedge \text{pre}(a) \subseteq F$ **do**
      if $a \notin q$ **then**
        $q.\text{insert}(a)$
      end if
    end for
  else
    $o \leftarrow q.\text{popBestOp}()$
    $O \leftarrow O \cup \{o\}$
    for $f \in \text{add}(e)$ **do**
      if $f \notin q$ **then**
        $q.\text{insert}(f)$
      end if
    end for
  end if
end while
return (F, O, I, G)

Este algoritmo presenta dos diferencias clave con respecto al anterior. La primera es que permite detenerse antes de procesar todos los elementos de la cola, utilizando un criterio de parada. La segunda es que introduce un orden de procesamiento definido para los elementos de la cola: se prioriza el procesamiento de los hechos sobre las acciones. Si no quedan hechos en la cola, la acción a procesar se selecciona mediante un criterio

heurístico, implementado en la función *popBestOp*. Esta función evalúa cada acción en la cola, y le asigna a cada una la probabilidad que tiene de formar parte del plan final. Una vez asignadas las probabilidades, la acción con mayor probabilidad será la procesada.

Para definir el criterio de parada, sabemos que al menos necesitamos que todos los hechos presentes en el objetivo (G) hayan sido instanciados. Es decir, necesitamos al menos que se cumpla $G \subseteq F$. Esto por sí solo no es suficiente para asegurar que todas las acciones necesarias para el plan hayan sido instanciadas. Sin embargo, con una heurística adecuada, que asigne consistentemente probabilidades altas a las acciones necesarias para resolver el problema, el algoritmo puede detenerse poco después de cumplir esta condición.

El desafío principal entonces está en definir la función *popBestOp*, de manera tal que distinga entre las acciones que serán necesarias para el plan final y aquellas que no, sin tener todavía el plan computado. A continuación, se presenta la idea general de la solución propuesta para definir esta función.

3.2. Función *popBestOp*

En la sección anterior, se presentó un algoritmo de grounding basado en un criterio heurístico diseñado para reducir aún más la cantidad de hechos y operadores instanciados. Este enfoque no se limita a instanciar todos los hechos y operadores alcanzables de forma relajada, sino que busca identificar y seleccionar únicamente aquellos que son relevantes para la construcción de un plan que resuelva el problema.

Esta estrategia hace uso de la función *popBestOp*, cuyo objetivo es, dado un problema de planificación y un conjunto de operadores asociados, seleccionar el operador que tiene mayor probabilidad de formar parte de un plan que resuelva el problema.

El foco central de este trabajo es dar una definición para *popBestOp*, utilizando un modelo de traducción basado en la arquitectura Transformer. En esta sección se explica la estrategia explorada para definir dicha función.

3.2.1. Idea general de la función

Sea $\Pi = \langle P, A, \Sigma^C, \Sigma^O, I, G \rangle$ un problema de planning, y π^+ un plan relajado del mismo. Llamamos *hechos relajados* al conjunto de hechos $p \in P^{\Sigma^C}$ que son verdaderos luego de aplicar el plan π^+ en el estado inicial I. Esto es, el conjunto de hechos $s \subseteq P^{\Sigma^C}$ tal que $I \xrightarrow{\pi^+} s$. Recordar que en la versión relajada de un problema, aplicar una acción no elimina hechos del estado. Por lo tanto, el conjunto de hechos al que se llega luego de aplicar el plan, está formado por todos los hechos que fueron agregados por cada acción del plan, junto con los hechos originalmente presentes en el estado inicial.

Tanto los planes relajados como sus correspondientes listas de hechos relajados pueden calcularse de manera eficiente utilizando herramientas como PowerLifted (Corrêa et al., 2020). Dado que es posible calcular eficientemente los planes y hechos relajados, surge la idea de utilizarlos como fuente de información para estimar la probabilidad de que cada operador forme parte de un plan que resuelva el problema. Esta propuesta se explora en Areces et al. (2023), donde la función *popBestOp* emplea un modelo de regresión logística entrenado con hechos relajados para asignar probabilidades a las acciones.

Cada dominio de planning tiene reglas específicas que definen cómo se representan sus hechos y acciones, lo cual está especificado en su descripción. Por ejemplo, volviendo al dominio de la aerolínea (presentado en el Código 2.1), tenemos hechos de la forma *(aterrizado ?avión ?aero)*, con *?avión* una variable que puede ser instanciada con objetos del tipo *AVION*, y *?aero* una variable que puede ser instanciada con objetos del tipo *AEROPUERTO*. Además, hay acciones como *(cargarCombustible ?avión)*, con sus precondiciones y efectos correspondientes, donde *?avión* es una variable que puede ser instanciada con objetos del tipo *AVION*.

Podemos pensar entonces en un "lenguaje" de los hechos y otro de las acciones de un dominio, cada uno con sus propias "estructuras gramaticales" que determinan las relaciones y restricciones entre sus elementos.

Aunque estas reglas gramaticales se especifican formalmente en la descripción del problema, también se reflejan en los planes y hechos relajados asociados a problemas en ese dominio. Además, los patrones en los planes y hechos relajados pueden evidenciar las relaciones entre precondiciones y postcondiciones de los operadores.

Por ejemplo, en el dominio presentado, tenemos que *(tieneCombustible avión1)* es una precondición necesaria para ejecutar la acción *(volar avión1 ezeiza ingAT)*. Al mismo tiempo, *(tieneCombustible avión1)* es una hecho que sólo puede ser alcanzado como postcondición de la acción *(cargarCombustible avión1)*. Por lo tanto, siempre que la acción *(volar avión1 ezeiza ingAT)* aparezca en un plan, debe haber sido precedida por la acción *(cargarCombustible avión1)*, a menos que *(tieneCombustible avión1)* ya esté presente en el estado inicial del problema.

De manera similar, si el hecho *(tieneCombustible avión1)* aparece en el conjunto de hechos relajados, se deduce que la acción *(cargarCombustible avión1)* debe haber sido incluida en el plan relajado correspondiente.

Este trabajo propone entonces el uso de un modelo de traducción automática para pasar de hechos a acciones; en particular para "traducir" los hechos relajados de un problema a un plan del mismo. El modelo idealmente aprendería estas reglas gramaticales y relaciones entre acciones a partir de los ejemplos de entrenamiento.

Una vez obtenido el plan generado por el modelo, podemos identificar qué acciones están presentes en él y, en función de esto, asignar una probabilidad a los diferentes operadores.

La función *popBestOp* operaría de la siguiente manera: primero calcula los hechos relajados del problema, luego los procesa mediante el modelo de traducción para obtener un plan. Con base en este plan, la función asigna una probabilidad a cada operador en la cola, dependiendo de su distancia respecto al plan generado. Finalmente, *popBestOp* selecciona y devuelve la acción con mayor probabilidad de formar parte del plan final.

3.2.2. Modelo de traducción

En este trabajo proponemos un modelo de traducción con arquitectura Transformer. El modelo toma como entrada los hechos relajados de un problema (computados previamente usando PowerLifted), y se espera que genere como salida una aproximación a un plan del problema. Se entrena el modelo para cada dominio específico, ya que cada uno tiene su propio vocabulario y reglas gramaticales.

Recordemos que dado un problema de planning Π y π^+ un plan relajado del mismo,

los hechos relajados son el conjunto de hechos que se obtiene luego de aplicar el plan relajado del problema. Esto es, el conjunto de hechos $s \subseteq P^{\Sigma^C}$ tal que $I \xrightarrow{\pi^+} s$. Es importante recordar que el modelo Transformer utiliza secuencias como entrada, no conjuntos, con lo cual es necesario obtener una secuencia de hechos a partir del conjunto de hechos relajados.

Para obtener una secuencia ordenada de los hechos relajados, consideramos dos opciones: la primera es ordenar los hechos alfabéticamente, y la segunda es ordenarlos según el momento en que fueron generados a lo largo del plan: los hechos agregados al aplicar la primera acción del plan aparecerán primero, seguidos por los generados por la segunda acción, y así sucesivamente. Denominaremos a estos conjuntos de hechos como hechos relajados en orden alfabético y hechos relajados en orden de aparición, respectivamente.

Así, los datos necesarios para entrenar el modelo son pares de hechos relajados y planes del problema. Para cada dominio específico D, tenemos un conjunto de problemas de entrenamiento $Train_D$. Para cada problema en este conjunto tenemos al menos un plan que resuelve el problema, y hechos relajados asociados a algún plan relajado del problema, en ambas versiones secuenciales. Para cada problema en $Train_D$, tomamos pares de la forma $(hechos\ relajados,\ plan)$ para cada plan que resuelve el problema. Estos pares forman el conjunto de entrenamiento del modelo. Se toma la decisión de armar un par por cada plan que resuelve el problema como una forma de aumentar la cantidad de datos de entrenamiento.

Sin embargo, es importante recordar que los hechos relajados corresponden a los hechos que se hacen verdaderos al aplicar un determinado plan relajado sobre el estado inicial. Esto presenta un desafío: podemos estar introduciendo ruido al modelo al emparejar hechos relajados con un plan que no le corresponde. Por ejemplo, supongamos que un problema tiene disponibles los objetos *avión1* y *avión2*, y que existen dos planes que resuelven la tarea, uno que usa *avión1* y otro que usa *avión2*. Si los hechos relajados se basan en el plan que utiliza *avión1*, pero los emparejamos con el plan que utiliza *avión2*, el vínculo entre los hechos relajados y el plan no será tan directo.

Recordemos que la primera etapa en el modelo de traducción es la tokenización de la entrada, y luego la salida del modelo asigna probabilidad a cada uno de los tokens del vocabulario de un tokenizador de salida. A continuación se explican los tokenizadores.

Tokenización de los datos

Recordemos que un Transformer usa dos tokenizadores: uno para la entrada y otro para la salida. El tokenizador de entrada se encarga de convertir el texto de origen, es decir, la entrada del modelo, en una secuencia de tokens que el modelo puede procesar. Por otro lado, el tokenizador de salida se utiliza para convertir los tokens predichos por el modelo de vuelta a texto legible.

Cada tokenizador trabaja con su propio vocabulario (los tokens que conoce), el cual es específico para el tipo de datos de entrada o salida. Así, el tokenizador de entrada debe ser capaz de tokenizar correctamente los hechos relajados, mientras que el tokenizador de salida debe manejar la conversión de los tokens generados por el modelo en un plan.

Además de los tokens correspondientes a cada lenguaje, ambos vocabularios incluyen tokens especiales como el de inicio de frase ('$\langle SOS \rangle$'), fin de frase ('$\langle EOS \rangle$'), token desconocido ('$\langle UNK \rangle$') y padding ('$\langle PAD \rangle$'). Cada tokenizador asigna un valor numérico a

cada token de su vocabulario, y durante el proceso de tokenización, los textos de entrada se convierten en secuencias de índices numéricos correspondientes a estos tokens. En la destokenización, los índices se transforman nuevamente en tokens.

Para determinar el vocabulario del tokenizador de entrada, vamos a tomar como tokens los elementos separados por un espacio (‘ ’) que encontramos en los hechos relajados de todos los pares de entrenamiento.

Analicemos un par de entrenamiento concreto del dominio de la aerolínea:

Hechos relajados:

(aterrizado avión1 ezeiza)
(tieneCombustible avión1)
(aterrizado avión1 ingAT)

Plan:

(cargarCombustible avión1)
(volar avión1 ezeiza ingAT)

Si este fuera el único ejemplo de entrenamiento de nuestro modelo, el tokenizador de entrada podría tener el siguiente vocabulario:

input_vocab:{‘⟨*SOS*⟩’ : 0, ‘⟨*EOS*⟩’ : 1, ‘⟨*UNK*⟩’ : 2, ‘⟨*PAD*⟩’ : 3, ‘*(*’ : 4, ‘*)*’ : 5,
‘*aterrizado*’ : 6, ‘*avión1*’ : 7, ‘*ezeiza*’ : 8, ‘*tieneCombustible*’ : 9,
‘*ingAT*’ : 10, ‘*cargarCombustible*’ : 11, ‘*volar*’ : 12}

Para tokenizar una entrada de la forma *(aterrizado avión1 ezeiza)*, con un largo máximo de secuencia de 10 tokens, el resultado sería [0, 4, 6, 7, 8, 5, 1, 3, 3, 3], donde:

- 0 corresponde a ‘⟨*SOS*⟩’, el token de inicio de secuencia.
- 4 corresponde al token‘*(*’, que marca el inicio de una acción.
- 6, 7 y 8 corresponden a los tokens ‘*aterrizado*’, ‘*avión1*’, y ‘*ezeiza*’, respectivamente.
- 5 corresponde al token ‘*)*’ que marca al final de una acción.
- 1 es el token de fin de secuencia ‘⟨*EOS*⟩’.
- Los tokens 3 son de relleno (‘⟨*PAD*⟩’) para completar la secuencia hasta el largo máximo de 10.

De manera análoga, se define el vocabulario para el tokenizador de salida, tomando los elementos separados por espacio encontrados en los planes de todos los pares de entrenamiento.

Acá aparece un problema: si el tokenizador no tiene un determinado token en su vocabulario, será reemplazado por el token '$\langle UNK \rangle$', lo que limita al modelo, ya que no podrá procesar ni aprender información específica sobre ese token desconocido. De manera similar, si un token no está presente en el vocabulario del tokenizador de salida, el modelo nunca podrá predecirlo, incluso si el significado o el patrón que desea representar está bien aprendido. Por lo tanto, garantizar que los vocabularios sean lo suficientemente amplios y representativos es esencial. Es fundamental que tanto el tokenizador de entrada como el de salida sean tan abarcativos como sea posible.

Además, no basta con que los tokens estén simplemente incluidos en el vocabulario; idealmente, deben estar también presentes en el conjunto de entrenamiento. Esto es fundamental porque, aunque un token esté en el vocabulario, si no aparece en los datos de entrenamiento, el modelo no podrá aprender información útil sobre cómo procesarlo o predecirlo correctamente en contextos futuros.

Supongamos que en los tokenizadores tenemos tanto '*avión1*' como '*avión2*', pero que el modelo se entrena solamente con ejemplos que usan '*avión1*'. Al presentarle un ejemplo idéntico a uno de entrenamiento pero reemplazando '*avión1*' por '*avión2*' podríamos suponer que podrá predecir correctamente el uso de '*avión2*' en vez de '*avión1*' en la traducción ya que ambos son objetos de tipo *AVION*, sólo que cambia el número. Esto no es así, el modelo no ve esta similitud sintáctica pues los va a tratar como tokens distintos. Sin embargo, si en el entrenamiento se le presentan algunos ejemplos que usan '*avión1*' y otros que usan '*avión2*', el modelo idealmente aprende esta semejanza semántica y lo refleja en el embedding de cada token.

Por lo tanto, necesitamos tokenizadores que tengan un vocabulario lo suficientemente completo y un conjunto de entrenamiento que contenga una variedad adecuada de tokens. Para lograrlo, proponemos una técnica de normalización de los problemas: los nombres de los objetos se formarán combinando el tipo del objeto con un número. Por ejemplo, el aeropuerto *ezeiza* se convertirá en *aeropuerto1* e *ingAT* en *aeropuerto2*. Esta normalización se aplica a todos los problemas de entrenamiento, asegurando que los tokens puedan ser reconocidos incluso si son nuevos o no han aparecido previamente en los datos de entrenamiento.

Es importante notar que en los problemas de planning, los nombres de las acciones (como *cargarCombustible*, *volar*, etc.) suelen aparecer en todos los planes, por lo que estos tokens estarán presentes durante el entrenamiento. Sin embargo, en los problemas de evaluación, podrían aparecer objetos que no se vieron en el entrenamiento, particularmente porque los ejemplos de entrenamiento suelen ser más simples y pequeños que los de evaluación. Por ejemplo, en los problemas de entrenamiento, podríamos usar *avión*i con $0 \leq i \leq 10$, pero en un problema de evaluación podría aparecer *avión125*, que no fue visto durante el entrenamiento.

Para solucionar esto, se propone que el vocabulario de los tokenizadores de entrada y de salida contenga los nombres de los objetos (esto es, '*avión*', '*aeropuerto*', etc), y los dígitos del 0 al 9 como tokens. Así, el tokenizador de entrada separa *(aterrizado avión1 aeropuerto2)*, en los tokens ['*(*', '*aterrizado*', '*avión*', '*1*', '*aeropuerto*', '*2*', '*)*'].

Este cambio permite que el tokenizador de entrada pueda procesar cualquier objeto de la forma *tipo_de_objeto número*, y a la vez el tokenizador de salida podría predecir cualquier objeto de este tipo en la salida. Esto aumenta la flexibilidad y la capacidad de generalización del modelo, permitiéndole manejar una mayor variedad de combinaciones

de objetos sin necesidad de que todos los tokens estén explícitamente presentes en el conjunto de entrenamiento. La desventaja de este enfoque es que se usarán más tokens para la misma entrada, con lo cual será necesario aumentar el largo máximo de secuencia para cubrir los mismos casos.

Habiendo explicado el modelo de traducción que usaremos, en particular la forma en que trataremos los datos de entrada y salida, en la siguiente sección se muestra su uso en el flujo de la función *popBestOp* y cómo obtener las probabilidades necesarias a partir de la salida del modelo.

3.2.3. El flujo completo de la función

A continuación, se explican las distintas etapas de la función *popBestOp*, y los cálculos realizados para pasar de la traducción generada por el Transformer a una asignación de probabilidades a cada operador que recibe como entrada la función.

Generación de las traducciones

Hay diversas formas de generar una traducción por medio del modelo. En este trabajo consideramos dos: *Greedy Decode* y *Beam Search*. Utilizaremos alguna o ambas estrategias para generar planes (traducciones) a partir de los hechos relajados.

Greedy Decode es una estrategia que, en cada paso de la generación, selecciona el token más probable para continuar la secuencia de salida. Este enfoque es simple y rápido, pero su principal limitación es que no explora otras posibles secuencias, y se pierde la oportunidad de considerar alternativas que podrían resultar en una mejor traducción final.

Por otro lado, *Beam Search* mantiene múltiples secuencias de traducción parciales en cada paso. En lugar de seleccionar únicamente el token más probable en cada iteración, este algoritmo conserva las n mejores secuencias según su probabilidad acumulada hasta el momento. De este modo, explora múltiples trayectorias simultáneamente y genera n posibles traducciones completas. El parámetro n, conocido como tamaño del beam, determina cuántas secuencias parciales se mantienen en cada paso del proceso.

Recordemos que la función *popBestOp* debe asignar probabilidades a una lista de operadores. A continuación, se describe cómo se hace uso de las traducciones generadas para poder asignar estas probabilidades.

Obtención de testigos

Una vez generadas las traducciones por medio del modelo, se crea un *Conjunto de Testigos*. Este conjunto está compuesto por los operadores que aparecen en las traducciones, considerando cada operador tantas veces como apariciones tenga en dichas traducciones y separándolos por tipos de operador (en el ejemplo de la aerolínea, un tipo de operador sería *volar*, otro *cargarCombustible*, etc.). El conjunto de testigos refleja los operadores que el modelo identifica como relevantes para el problema, basándose en las traducciones generadas.

Idealmente, la traducción obtenida por el modelo es un plan que soluciona el problema. En este caso, los operadores presentes en el conjunto de testigos serían aquellos que efectivamente aparecen en un plan. Sin embargo, el modelo puede cometer errores y

generar traducciones que no constituyan planes correctos. Por lo tanto, aunque el conjunto de testigos proporciona información útil sobre los operadores relevantes, no podemos asumir que sea completamente preciso.

A la hora de asignar probabilidades a la lista de operadores, consideramos la información contenida en el conjunto de testigos como una guía. Una posibilidad inicial sería asignar probabilidad 1 a los operadores presentes en el conjunto de testigos y probabilidad 0 a los que no están presentes. Sin embargo, esta estrategia es demasiado rígida y no contempla la posibilidad de que el modelo haya generado una traducción incorrecta.

Por lo tanto, debemos usar de forma más flexible el conjunto de testigos, para poder manejar posibles errores por parte del modelo. En lugar de determinar únicamente si un operador está o no presente en el conjunto de testigos, introducimos el concepto de *distancia* entre un operador y el conjunto de testigos.

Esta distancia proporciona una medida de qué tan cercano o relevante es un operador con respecto a los operadores identificados en la traducción generada por el modelo. A continuación, se presentan distintas formas en que puede definirse esta distancia.

Cálculo de Distancias

Recordemos que, dado un problema, la función *popBestOp* se encarga de asignar a cada operador de una lista la probabilidad que tiene de formar parte de un plan que resuelva el problema. Para calcular esta probabilidad, primero determinamos la distancia entre cada operador y los testigos del mismo tipo en el conjunto de testigos. Para esto, primero se calcula la *distancia individual* entre el operador operador y cada testigo del mismo tipo, y luego a partir de las distancias individuales se calcula la *distancia final* entre el operador y el conjunto completo de testigos del mismo tipo.

La distancia individual entre un operador y un testigo concreto (del mismo tipo que el operador considerado), se define como el número de tokens en los que difieren ambos, dividido por la cantidad total de tokens del operador (para normalizar la comparación entre operadores de diferentes tamaños). Esta distancia representa que tan diferente es un operador a un testigo determinado.

Recordemos que el tokenizador de salida toma como tokens separados el tipo de cada objeto y cada dígito. Así, el operador *(volar avión1 aeropuerto57 aeropuerto3)*, se tokenizaría como [‘*(*’, ‘*volar*’, ‘*avión*’, ‘*1*’, ‘*aeropuerto*’, ‘*5*’, ‘*7*’, ‘*aeropuerto*’, ‘*3*’, ‘*)*’].

Esta forma de tokenizar no es útil en este momento, ya que para poder comparar dos tokens del mismo tipo, necesitamos que tengan la misma cantidad de tokens. Sin embargo, al considerar cada dígito como su propio token, esto no se cumple. Entonces, en esta etapa para tokenizar vamos a separar los objetos en su tipo y su número, sin importar la cantidad de dígitos. Así, por ejemplo *avión123* se tokeniza como [‘*avión*’, ‘*123*’] y no como [‘*avión*’, ‘*1*’, ‘*2*’, ‘*3*’].

Ahora, dado un operador y un conjunto de testigos, consideramos dos estrategias para calcular la distancia final entre un operador y el conjunto de testigos del tipo de el operador. Esta distancia representa que tan diferente es el operador al conjunto de testigos.

La primera estrategia, que llamaremos *distancia mínima*, consiste en calcular la distancia individual entre el operador y cada testigo del mismo tipo, y tomar la menor de estas distancias como la distancia final. Esto asume que la mayor similitud (es decir, la menor distancia) entre el operador y cualquier testigo del conjunto es la más representativa.

La segunda estrategia, que llamaremos *distancia promedio*, consiste en calcular la distancia individual entre el operador y cada testigo del mismo tipo, sumar todas estas distancias individuales y dividir el resultado por la cantidad total de testigos del tipo correspondiente. Este enfoque considera la relación del operador con todo el conjunto de testigos, proporcionando una medida más global de disimilitud.

Una diferencia clave entre estas dos estrategias radica en cómo manejan la redundancia de los operadores en múltiples traducciones. Si se generan múltiples traducciones y un operador aparece en más de una de ellas, la estrategia de distancia promedio lo reflejará dado que se promedian múltiples valores. Este comportamiento es particularmente útil, ya que la repetición de un operador en varias traducciones podría interpretarse como una señal de mayor confianza del modelo en que dicho operador pertenece al plan final.

El cálculo de distancias explicado se basa exclusivamente en la salida del modelo. Sin embargo, también se dispone de información adicional proveniente de los hechos relajados generados por PowerLifted. Si bien los hechos relajados son la entrada para el modelo (y, por ende, influyen en la salida), es posible aprovecharlos directamente. A continuación, se describe una estrategia que incorpora explícitamente los hechos relajados.

Porcentaje de objetos no presentes en los hechos relajados

Sea x un operador o un conjunto de hechos, definimos $obj(x)$ como el conjunto de objetos presentes en x. Por ejemplo,

$$obj((\ volar\ avión1\ aeropuerto2\ aeropuerto1)) = \{avión1, aeropuerto2, aeropuerto1\}.$$

Ahora, dado un conjunto de hechos h y un operador o, definimos el porcentaje de objetos de o que no están presentes en h como

$$porc\_obj(o,h) = \frac{|obj(o) \setminus obj(h)|}{|obj(o)|} \tag{3.1}$$

$$= \frac{\text{Cantidad de objetos de o no presentes en h}}{\text{Cantidad total de objetos en o}} \tag{3.2}$$

Si $porc\_obj(o,h) = 0$, significa que todos los objetos de o están presentes en h. Por el contrario, un valor cercano a 1 indica que la mayoría de los objetos de o no están en h.

Notemos que si los objetos de un operador no aparecen en los hechos relajados de un problema, podríamos pensar que el operador no será parte del plan asociado a esos hechos relajados. Esto se debe a que, en general, las precondiciones y postcondiciones de un operador dependen de los objetos que lo componen. Si un operador pertenece al plan final, sus precondiciones y postcondiciones por lo general están reflejadas en los hechos relajados del problema.

Es por esto que surge la idea de utilizar el porcentaje de objetos no presentes en los hechos relajados como otra forma de medir qué tan relevante será un operador para el plan de un problema. Cuanto menor sea este porcentaje, más probable será que el operador forme parte de un plan que resuelve el problema.

Una vez calculada la distancia final y/o el porcentaje de objetos no presentes en los hechos relajados para cada operador en la lista de entrada de $popBestOp$, calculamos la probabilidad final asignada a cada uno de ellos. Este cálculo se explica a continuación.

Cálculo de probabilidades

Para calcular la probabilidad asignada a cada operador, se puede trabajar con uno o ambos valores calculados previamente: la distancia final o el porcentaje de objetos no presentes en los hechos relajados. Si se utiliza solo uno de estos valores, se toma directamente como base para los cálculos que se explican a continuación. En caso de que se usen ambos valores, primero se calcula el promedio entre ellos para obtener una medida combinada. Este valor final, ya sea individual o promedio, representa cuán diferente es cada operador respecto al plan esperado para los hechos relajados del problema.

Dado que una menor distancia implica una mayor similitud con el plan esperado, se calcula el opuesto del valor obtenido (ya sea la distancia, el porcentaje, o su promedio). Este cálculo convierte las distancias en puntajes de similitud: a menor distancia, mayor será el puntaje de similitud asignado a la acción.

Para transformar los puntajes de similitud en probabilidades, se utiliza la función softmax. Esta técnica toma los puntajes calculados para todos los operadores en la cola y los convierte en valores que se pueden interpretar como probabilidades, es decir, números entre 0 y 1 que suman exactamente 1.

La función softmax asigna mayor probabilidad a los operadores con puntajes de similitud más altos, manteniendo la relación proporcional entre ellos. De esta manera, cada operador recibe un valor que refleja su relevancia relativa frente a los demás operadores.

Finalmente, el operador con la mayor probabilidad es seleccionada como la próxima a ejecutar por la función $popBestOp$. Esto garantiza que se prioricen los operadores más relevantes según la métrica seleccionada o combinada.

Con esto concluye la propuesta de este trabajo para el cálculo de probabilidades por la función $popBestOp$. A continuación, se presenta la métrica utilizada para evaluar el desempeño de esta estrategia.

3.3. Métrica para evaluar la solución propuesta: PUO

Dado un dominio de planning, podemos considerar el conjunto completo de operadores asociados a dicho dominio. Este conjunto puede dividirse, para un problema específico, en dos subconjuntos: los *Buenos Operadores*, que son aquellos que aparecen en algún plan que resuelve el problema; y su complemento, los *Malos Operadores*, que no aparecen en ningún plan que resuelve el problema.

Este conjunto completo de operadores del dominio, puede ordenarse de forma aleatoria para obtener una lista y pasarla como entrada a la función $popBestOp$. La función asigna a cada operador del dominio una probabilidad, y a partir de estas probabilidades, se puede reordenar la lista de operadores en orden decreciente según su probabilidad. En esta lista ordenada, el operador con mayor probabilidad asignada aparece en primer lugar.

Dada esta lista de operadores ordenados según su probabilidad, podemos identificar el menor índice tal que, considerando los operadores desde el inicio hasta dicho índice, sea posible construir un plan que resuelva el problema (no necesariamente utilizando todos los operadores). Idealmente, $popBestOp$ asignará probabilidades altas a los Buenos Operadores y bajas a los malos, lo que resultará en un índice bajo.

Ahora, en el Algoritmo 2 (donde se muestra cómo obtener la versión STRIPS de un problema en PDDL siguiendo un proceso de grounding con criterio heurístico), se van

agregando operadores a la versión STRIPS del problema a medida que son seleccionados por *popBestOp*. Recordemos que la idea del algoritmo es obtener una versión STRIPS del problema con la menor cantidad posible de operadores innecesarios para encontrar un plan del problema. Entonces, si queremos agregar operadores de forma tal que permitan formar un plan que resuelva el problema, es suficiente con agregar los operadores hasta el índice encontrado previamente.

Así, la métrica usada para medir el desempeño de *popBestOp*, será el *Porcentaje de Operadores No Instanciados*, o *PUO* por sus siglas en inglés (*Percentage of Ungrounded Operators*), que se define como la proporción de operadores con probabilidad menor a la del índice mínimo, sobre el total de operadores en la lista.

Esta métrica evalúa la eficiencia de *popBestOp* para priorizar operadores relevantes: un PUO cercano a 100 % indica que la función asigna mayores probabilidades a los Buenos Operadores, colocándolos en las primeras posiciones de la lista y asigna probabilidades más bajas al resto de los operadores, que no serán instanciados.

Para poder implementar esta idea, vamos a tomar un subconjuntos finitos de buenos y Malos Operadores, y la unión de estos será un subconjunto del conjunto de todos los operadores del dominio. No podemos trabajar con los conjuntos enteros de Buenos y Malos Operadores porque si bien son conjuntos finitos, pueden ser muy grandes y el costo computacional sería muy alto.

Para el subconjunto de Buenos Operadores, seleccionamos un plan que resuelve el problema y tomamos todos los operadores de este plan como Buenos Operadores. En cuanto al subconjunto de Malos Operadores, buscamos que sea lo suficientemente grande y diverso para representar adecuadamente el conjunto original. Para lograr esto, se entrenó un modelo que genera embeddings para los operadores. Estos embeddings tienen la propiedad de asignar posiciones similares en el espacio vectorial a operadores parecidos.

A partir de estos embeddings, se seleccionaron hasta 50000 Malos Operadores asegurando que la distancia entre cada par de operadores seleccionados superara un umbral predefinido. Este criterio garantiza que los operadores elegidos sean lo más diversos posible, ya que al priorizar aquellos que están lejos entre sí en el espacio vectorial, capturamos una mayor variedad de características de los operadores. La construcción de los subconjuntos de buenos y Malos Operadores ya había sido realizada en Areces et al. (2023), y esta información fue usada en este trabajo.

Entonces, para evaluar *popBestOp*, se le asignan probabilidades a todos los operadores del conjunto dado por la unión de los conjuntos finitos de buenos y Malos Operadores. Luego, estos se ordenan en orden decreciente según sus probabilidades y se identifica el menor índice tal que se puede formar un plan que resuelva el problema usando operadores con probabilidad mayor o igual a la del operador en dicho índice. Finalmente, se calcula el PUO como la fracción de operadores con probabilidad menor a la asignada al operador en el índice encontrado con respecto al total de operadores.

Idealmente, *popBestOp* asignará las probabilidades más altas a los Buenos Operadores, de manera que el índice mínimo incluiría sólo Buenos Operadores y el PUO será cercano a 100, esto reflejaría un desempeño óptimo.

3.4. Ejemplo de selección de acción con *popBestOp*

Analicemos un ejemplo dentro del dominio de la aerolínea. Supongamos que los hechos relajados son:

(aterrizado avión1 aeropuerto1),
(tieneCombustible avión1),
(aterrizado avión1 aeropuerto3).

El conjunto de testigos generados es:

(cargarCombustible avión1),
(cargarCombustible avión1),
(cargarCombustible avión2),
(volar avión1 aeropuerto2 aeropuerto1),
(volar avión1 aeropuerto1 aeropuerto3).

Y la cola de acciones pasada a *popBestOp* es:

(cargarCombustible avión1),
(volar avión1 aeropuerto1 aeropuerto2).

Para cada acción en la cola de acciones, calculamos las distancias individuales con los testigos del mismo tipo. Recordemos que esto es cantidad de tokens distintos sobre total de tokens, así tenemos:

$$d(\textit{(cargarCombustible avión1), (cargarCombustible avión1)}) = 0,$$
$$d(\textit{(cargarCombustible avión1), (cargarCombustible avión1)}) = 0,$$
$$d(\textit{(cargarCombustible avión1), (cargarCombustible avión2)}) = \frac{1}{4},$$
$$d(\textit{(volar avión1 aeropuerto1 aeropuerto2), (volar avión1 aeropuerto2 aeropuerto1)}) = \frac{2}{6},$$
$$d(\textit{(volar avión1 aeropuerto1 aeropuerto2), (volar avión1 aeropuerto1 aeropuerto3)}) = \frac{1}{6}.$$

Una vez calculadas las distancias individuales para cada acción en la cola, se computan las distancias finales. Se presentan los resultados para ambas estrategias explicadas, distancia mínima y distancia promedio. Si utilizamos la estrategia de distancia mínima se obtiene:

$$\text{Para } \textit{(cargarCombustible avión1)} = \text{mín}\left\{0, 0, \frac{1}{4}\right\} = 0.$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = \text{mín}\left\{\frac{2}{6}, \frac{1}{6}\right\} = \frac{1}{6}.$$

En cambio, con la estrategia de distancia promedio los resultados son:

$$\text{Para } \textit{(cargarCombustible avión1)} = \frac{0+0+\frac{1}{4}}{3} = \frac{1}{12}$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = \frac{\frac{2}{6}+\frac{1}{6}}{2} = \frac{1}{4}$$

Luego calculamos el porcentaje de objetos que no están presentes en los hechos relajados para cada acción. Para la acción *(cargarCombustible avión1)*, tenemos:

$$obj(\text{acción}) = \{\textit{avión1}\},$$
$$obj(\text{hechos relajados}) = \{\textit{avión1}, \textit{aeropuerto1}, \textit{aeropuerto3}\}.$$

En este caso, el único objeto del acción está presente en los hechos relajados. Asi:

$$porc\_obj(\textit{(cargarCombustible avión1)}, \text{ hechos relajados}) = \frac{0}{1} = 0$$

Ahora, calculamos el porcentaje de objetos que no están presentes en los hechos relajados para la acción *(volar avión1 aeropuerto1 aeropuerto2)*. Tenemos:

$$obj(\text{acción}) = \{\textit{avión1}, \textit{aeropuerto1}, \textit{aeropuerto2}\},$$
$$obj(\text{hechos relajados}) = \{\textit{avión1}, \textit{aeropuerto1}, \textit{aeropuerto3}\}.$$

El objeto *aeropuerto2* no está en los hechos relajados, y hay 3 objetos en la acción. Así:

$$porc\_obj(\textit{(volar avión1 aeropuerto1 aeropuerto2)}, \text{ hechos relajados}) = \frac{1}{3}$$

Una vez calculada la distancia final, y el porcentaje de objetos no presentes en los hechos relajados para cada acción, tomamos el promedio entre ambos para obtener la distancia final entre la acción y el conjunto de testigos. Consideramos por separado el caso en el que se usa la distancia mínima y la distancia promedio como distancia final. Si se usa la distancia mínima como distancia final, tenemos:

$$\text{Para } \textit{(cargarCombustible avión1)} = \frac{0+0}{2} = 0.$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = \frac{\frac{1}{6}+\frac{1}{3}}{2} = \frac{1}{4}$$

Por otro lado, si se usa la distancia promedio, la distancia final está dada por:

$$\text{Para } \textit{(cargarCombustible avión1)} = \frac{\frac{1}{12}+0}{2} = \frac{1}{24}.$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = \frac{\frac{1}{4}+\frac{1}{3}}{2} = \frac{7}{24}.$$

Una vez obtenida la distancia final, se calculan las probabilidades para cada acción aplicando softmax sobre el opuesto de las distancias finales. Si se usa la distancia mínima como distancia final, la probabilidad final asignada a cada acción es:

$$\text{Para } \textit{(cargarCombustible avión1)} = 0{,}5622.$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = 0{,}4378.$$

En cambio, usando la distancia promedio como distancia final, las probabilidades asignadas son:

$$\text{Para } \textit{(cargarCombustible avión1)} = 0{,}5645.$$
$$\text{Para } \textit{(volar avión1 aeropuerto1 aeropuerto2)} = 0{,}4355.$$

Con ambas estrategias, *(cargarCombustible avión1)* es la acción con mayor probabilidad. Esta accón es la seleccionada por la función *popBestOp* como la más relevante para el problema. Es importante notar que, si bien se presentaron ambas estrategias para ejemplificar, en la aplicación se utilizará una sola de ellas.

Capítulo 4

Implementación del modelo de traducción

Dado que la arquitectura Transformer ha demostrado ser altamente efectiva para problemas de traducción automática, la elegimos para nuestro enfoque. Nuestra implementación utiliza una arquitectura Transformer clásica, desarrollada en Python con la librería PyTorch (Paszke et al., 2019). En este capítulo se describe cómo se entrenó el modelo de traducción con arquitectura Transformer. Además, se muestra el código de implementación de este modelo, asociandolo con la teoría de esta arquitectura presentada en la sección 2.6.

El código presentado en esta sección puede encontrarse en este repositorio.

4.1. Entrenamiento del modelo

En un modelo Transformer, el Decoder recibe como entrada tanto la representación generada por el Encoder como la traducción parcial generada hasta el momento, así al buscar la palabra para una determinada posición en la secuencia, tiene en cuenta el contexto brindado por todas las palabras anteriores. Durante el entrenamiento, en lugar de usar la traducción parcial generada por el modelo, se utiliza la traducción completa del objetivo, tomada del par de entrenamiento. Esto permite que el modelo asigne probabilidades a cada token de la secuencia de salida basándose en los tokens correctos de las posiciones anteriores, lo que se conoce como *teacher forcing* (Williams and Zipser, 1989).

Este enfoque evita que el modelo acumule errores al depender únicamente de sus propias predicciones durante el proceso de entrenamiento. Además, permite que las probabilidades para todos los tokens de la secuencia de salida se calculen simultáneamente, lo que contrasta con el proceso de inferencia, donde el modelo predice un token a la vez de manera secuencial, utilizando los tokens previamente generados como entrada para predecir los siguientes.

Recordemos que la salida del Transformer es una asignación de probabilidades para cada token del vocabulario de salida. Durante la inferencia, el modelo predice los tokens de forma secuencial, agregando uno en cada paso. En cambio, durante el entrenamiento, calcula una distribución de probabilidad para cada token en todas las posiciones de la secuencia de salida simultáneamente. Idealmente, querríamos que, en la i-ésima posición de la secuencia de salida, el modelo le asigne probabilidad 1 al token correspondiente a la

i-ésima posición de la traducción objetivo, y 0 a los demás tokens. La función de pérdida utilizada, la *entropía cruzada* (*cross-entropy*), mide para cada posición de la secuencia de salida la distancia entre las probabilidades asignadas a cada token por el modelo y la distribución esperada, que sería un vector que tiene 0 en todos los elementos, salvo en el correspondiente al token correcto, para el cual tiene un valor de 1.

Formalmente, para un par de entrenamiento (x, y), con $y = [y_1, \ldots, y_{max\_seq}]$ siendo la secuencia de salida deseada, y $o \in \mathbb{R}^{max\_seq \times vocab\_size}$ la salida del modelo al pasarle x como entrada, la entropía cruzada se define como:

$$L = -\sum_{i=1}^{max\_seq} \log(o_{i,id(y_i)})$$

Aquí, $o_{i,j}$ representa la probabilidad asignada al token j en la posición i de la salida. Idealmente, queremos que $o_{i,j} = 1$ si $id(y_i) = j$ y $o_{i,j} = 0$ en caso contrario.

El modelo se entrenó durante 50 épocas, utilizando *Adam* (Kingma and Ba, 2015) como optimizador. Adam ajusta los parámetros del modelo utilizando no solo el gradiente actual, sino también promedios de gradientes pasados y sus velocidades de cambio, lo que lo hace más robusto y eficiente en comparación con el descenso de gradiente estándar.

Para optimizar los hiperparámetros del modelo, se utilizó el optimizador *Optuna* (Akiba et al., 2019), que usa el método *TPE* (*Tree-structured Parzen Estimator*) para explorar configuraciones prometedoras de hiperparámetros basándose en distribuciones probabilísticas obtenidas de los experimentos previos. Se ajustaron el tamaño del embedding, el número de bloques de Encoder y Decoder, el número de cabezales de atención y la tasa de aprendizaje, con el objetivo de maximizar el desempeño en el conjunto de validación.

4.2. Construcción de los componentes y el modelo

La función que se presenta a continuación es la encargada de construir los distintos componentes necesarios para el modelo, y luego los utiliza para crear el modelo en sí.

```
''' Función que construye el modelo Transformer
  src_vocab_size: tamaño del vocabulario de la entrada
  tgt_vocab_size: tamaño del vocabulario de la salida
  src_seq: tamaño máximo de la secuencia de entrada
  tgt_seq: tamaño máximo de la secuencia de salida
  d_model: tamaño del embedding
  N_Encoder: número de bloques de Encoder
  N_Decoder: número de bloques de Decoder
  h: número de cabezas de atención
  dropout: probabilidad de dropout
  d_ff: dimensión de la capa feed forward '''

def build_transformer(src_vocab_size: int, tgt_vocab_size: int, src_seq: int,
    tgt_seq: int, d_model: int=512, N_Encoder: int=6, N_Decoder: int=6, h:
    int=8, dropout: float=0.1, d_ff: int=2048) -> Transformer:

  # Embeddings para las entradas del Encoder y Decoder
  src_embed = InputEmbeddings(d_model, src_vocab_size)
  tgt_embed = InputEmbeddings(d_model, tgt_vocab_size)

```

```
  # Embeddings posicionales para las entradas del Encoder y Decoder
  src_pos = PositionalEncoding(d_model, src_seq, dropout)
  tgt_pos = PositionalEncoding(d_model, tgt_seq, dropout)

  # Creación de bloques de Encoder y luego el Encoder completo
  Encoder_blocks = []
  for _ in range(N_Encoder):
    Encoder_self_attention_block = MultiHeadAttentionBlock(d_model, h, dropout)
    feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
    Encoder_block = EncoderBlock(d_model, Encoder_self_attention_block,
        feed_forward_block, dropout)
    Encoder_blocks.append(Encoder_block)
  Encoder = Encoder(d_model, nn.ModuleList(Encoder_blocks))

  # Creación de bloques de Decoder y luego el Decoder completo
  Decoder_blocks = []
  for _ in range(N_Decoder):
    Decoder_self_attention_block = MultiHeadAttentionBlock(d_model, h, dropout)
    Decoder_cross_attention_block = MultiHeadAttentionBlock(d_model, h, dropout)
    feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
    Decoder_block = DecoderBlock(d_model, Decoder_self_attention_block,
        Decoder_cross_attention_block, feed_forward_block, dropout)
    Decoder_blocks.append(Decoder_block)
  Decoder = Decoder(d_model, nn.ModuleList(Decoder_blocks))

  # Creación de la capa de proyección
  projection_layer = ProjectionLayer(d_model, tgt_vocab_size)

  # Creación del Transformer
  transformer = Transformer(Encoder, Decoder, src_embed, tgt_embed, src_pos,
      tgt_pos, projection_layer)

  # Inicialización de los parámetros
  for p in transformer.parameters():
    if p.dim() > 1:
      nn.init.xavier_uniform_(p)

  return transformer
```

La función anterior crea el modelo de tranducción con todos sus bloques y etapas. Siguiendo la teoría, se necesita:

- Una función que pase del texto de entrada a un embedding.
- Una función que construya el embedding posicional para la entrada del Encoder.
- Una función que pase del texto en el idioma de salida a un embedding.
- Una función que construya el embedding posicional para la entrada del Decoder.
- El Encoder, compuesto por $N_{Encoder}$ bloques de Encoder, cada uno con atención con múltiples cabezales y una red neuronal feed forward fully connected.
- El Decoder, compuesto por $N_{Decoder}$ bloques de Decoder, cada uno con atención de múltiples cabezales simple y cruzada, y una red neuronal feed forward fully connected.

- Una fución de proyección para pasar la salida del Decoder a probabilidades para cada token del vocabulario de salida.

Teniendo esto en cuenta, se crean los distintos bloques del Transformer.

En la línea 19 se define el objeto encargado de construir los embeddings de entrada para el Encoder. Este asignará un vector único de dimensión `d_model` a cada uno de los `src_vocab_size` tokens del vocabulario de entrada. De manera similar, en la línea 20 se define el objeto que generará los embeddings para las entradas del Decoder.

En las líneas 23 y 24 se definen los objetos responsables de calcular los embeddings posicionales para el Encoder y el Decoder, respectivamente. El embedding posicional del Encoder será una matriz de dimensión `src_seq` $\times$ `d_model`, donde `src_seq` representa la longitud máxima permitida para las secuencias de entrada en tokens.

A continuación, se construyen los bloques del Encoder y del Decoder. Cada bloque de Encoder incluye un mecanismo de atención con `h` cabezales aplicado sobre entradas de tamaño `d_model`. Luego, una red neuronal feed forward fully connected transforma las entradas de dimensión `d_model` a `d_ff` y luego de nuevo a `d_model`. Por otra parte, cada bloque de Decoder incluye un mecanismo de atención con `h` cabezales aplicado sobre sus propias entradas, seguido de una atención cruzada con las salidas del Encoder. Finalmente, también incluye una red feed forward fully connected de dimensión `d_ff`, con dropout aplicado en cada componente. Se construyen `N_Encoder` bloques de Encoder y `N_Decoder` bloques de Decoder.

Por último, se define la capa de proyección, que convierte los vectores de tamaño `d_model` a vectores de tamaño `tgt_vocab_size`. Esto permite asignar un valor a cada token del vocabulario de salida.

En las siguientes secciones, se presenta el código de cada uno de los componentes utilizados y del modelo en sí.

4.3. Preprocesamiento de la entrada

En esta sección se presenta el código de los componentes encargados de convertir la secuencia de tokens de entrada en representaciones numéricas que el modelo pueda procesar. Esto incluye la generación de embeddings que representan el significado de los tokens y la incorporación de información posicional mediante embeddings posicionales, lo que permite al modelo interpretar el significado de cada token teniendo en cuenta su posición en la secuencia total.

```
1  class InputEmbeddings(nn.Module):
2  def __init__(self, d_model: int, vocab_size: int):
3    super().__init__()
4    self.d_model = d_model
5    self.vocab_size = vocab_size
6    self.embedding = nn.Embedding(vocab_size, d_model)
7
8  def forward(self, x): # (batch, seq) -> (batch, seq, d_model)
9    # Se multiplica por la raíz cuadrada para normalizar los embeddings
10   return self.embedding(x) * math.sqrt(self.d_model)
```

La clase `InputEmbeddings` se encarga de pasar los tokens de entrada a un embedding de dimensión `d_model`. Al pasarle como entrada un lote con `batch` entradas, cada una

con `seq` tokens, devuelve una matriz donde a cada token de cada entrada le asigna un embedding de dimensión `d_model`; es decir devuelve una matriz de dimensión `batch` × `seq` × `d_model`.

```

class PositionalEncoding(nn.Module):
  def __init__(self, d_model: int, seq: int, dropout: float) -> None:
    super().__init__()
    self.d_model = d_model
    self.seq = seq
    self.dropout = nn.Dropout(dropout)

    # Crea una matriz de tamaño (seq, d_model)
    pe = torch.zeros(seq, d_model)

    position = torch.arange(0, seq, dtype=torch.float).unsqueeze(1) # (seq, 1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() *
        (-math.log(10000.0) / d_model)) # (d_model / 2)

    # Aplica la fórmula con seno a las posiciones pares
    pe[:, 0::2] = torch.sin(position * div_term) # sin(position * (10000 ** (2i
        / d_model)))

    # Aplica la fórmula con coseno a las posiciones impares
    pe[:, 1::2] = torch.cos(position * div_term) # cos(position * (10000 ** (2i
        / d_model)))

    # Permite procesamiento de lotes
    pe = pe.unsqueeze(0) # (1, seq, d_model)

    # Lo guarda como un buffer para que no se actualice en el entrenamiento
    self.register_buffer('pe', pe)

  def forward(self, x):
    # Agrega a la entrada el embedding posicional
    x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False) # (batch, seq,
        d_model)
    # dropout para regularizar
    return self.dropout(x)
```

La clase `PositionalEncoding` se encarga de generar el embedding posicional para las entradas del modelo y agregarlo a los embeddings de entrada. Utiliza funciones seno y coseno para obtener una representación única para cada token en cada posición en la secuencia. Esta representación se suma a los embeddings de los tokens para que el modelo tenga información tanto del contenido del token como de su posición. Finalmente, se aplica dropout para regularizar el modelo durante el entrenamiento.

4.4. Componentes auxiliares

A continuación, se muestra el código de diversos componentes que formarán parte de cada bloque de Encoder y Decoder: la conexión residual, la normalización, y la red neuronal feed forward fully connected.

```python
class LayerNormalization(nn.Module):

  def __init__(self, features: int, eps:float=10**-6) -> None:
    super().__init__()
    self.eps = eps
    # alpha y bias son parametros aprendibles
    self.alpha = nn.Parameter(torch.ones(features))
    self.bias = nn.Parameter(torch.zeros(features))

  def forward(self, x):
    # x: (batch, seq, hidden_size)

    mean = x.mean(dim = -1, keepdim = True) # (batch, seq, 1)

    std = x.std(dim = -1, keepdim = True) # (batch, seq, 1)

    return self.alpha * (x - mean) / (std + self.eps) + self.bias
```

La clase `LayerNormalization` implementa la normalización que ajusta las activaciones restando su media y dividiendo por su desviación estándar a lo largo de la dimensión de características. Luego, escala y desplaza la salida mediante los parámetros aprendibles `alpha` y `bias`.

```python
  class ResidualConnection(nn.Module):

      def __init__(self, features: int, dropout: float) -> None:
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.norm = LayerNormalization(features)

      def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

La clase `ResidualConnection` se encarga de realizar la normalización y sumarlo a la entrada original.

```python
class FeedForwardBlock(nn.Module):

  def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
    super().__init__()
    self.linear_1 = nn.Linear(d_model, d_ff)
    self.dropout = nn.Dropout(dropout)
    self.linear_2 = nn.Linear(d_ff, d_model)

  def forward(self, x):
    # (batch, seq, d_model) -> (batch, seq, d_ff) -> (batch, seq, d_model)
    return self.linear_2(self.dropout(torch.relu(self.linear_1(x))))
```

La clase `FeedForwardBlock` es una red neuronal feed forward fully connected. Simplemente tiene una capa lineal para pasar cada embedding de cada token de la entrada de dimensión `d_model` a dimensión `d_ff`, aplica *ReLU* como activación no lineal y vuelve a la dimensión `d_model`. El uso de la función no lineal *ReLU* permite que la red aprenda patrones más complejos.

4.5. Atención con múltiples cabezales

A continuación, se presenta el componente que implementa la atención con múltiples cabezales. Será usado en la atención simple del Encoder, y en la cruzada del Decoder.

```python
class MultiHeadAttentionBlock(nn.Module):
  def __init__(self, d_model: int, h: int, dropout: float) -> None:
    super().__init__()
    self.d_model = d_model # Dimensión del embedding de entrada
    self.h = h # Cantidad de cabezales de atención
    assert d_model % h == 0, "d_model is not divisible by h"
    self.d_k = d_model // h # Dimensión del vector visto por cada cabezal
    self.w_q = nn.Linear(d_model, d_model, bias=False) # Matriz de consulta
    self.w_k = nn.Linear(d_model, d_model, bias=False) # Matriz de clave
    self.w_v = nn.Linear(d_model, d_model, bias=False) # Matriz de valor
    self.w_o = nn.Linear(d_model, d_model, bias=False) # Matriz de salida
    self.dropout = nn.Dropout(dropout)

  @staticmethod
  def attention(query, key, value, mask, dropout: nn.Dropout):
    d_k = query.shape[-1]

    # Asigna el puntaje de atención a cada par de palabras en cada cabezal
    attention_scores = (query @ key.transpose(-2, -1)) / math.sqrt(d_k)

    if mask is not None: # Aplica la máscara (si corresponde)
      attention_scores.masked_fill_(mask == 0, -1e9) # Asigna valores muy bajos
    attention_scores = attention_scores.softmax(dim=-1) # (batch, h, seq, seq)
    if dropout is not None:
      attention_scores = dropout(attention_scores)

    # Multiplicación de los puntajes de atención por los valores
    return (attention_scores @ value), attention_scores

  def forward(self, q, k, v, mask):
    # Obtiene el vector de consulta, clave y valor para cada token
    query = self.w_q(q) # (batch, seq, d_model) --> (batch, seq, d_model)
    key = self.w_k(k) # (batch, seq, d_model) --> (batch, seq, d_model)
    value = self.w_v(v) # (batch, seq, d_model) --> (batch, seq, d_model)

    # Separa los vectores de consulta, clave y valor para cada cabeza
    # (batch, seq, d_model) --> (batch, seq, h, d_k) --> (batch, h, seq, d_k)
    query = query.view(query.shape[0], query.shape[1], self.h,
        self.d_k).transpose(1, 2)
    key = key.view(key.shape[0], key.shape[1], self.h, self.d_k).transpose(1, 2)
    value = value.view(value.shape[0], value.shape[1], self.h,
        self.d_k).transpose(1, 2)

    # Atención sobre cada cabezal y combina el resultado de todos los cabezales
    # (batch, h, seq, d_k) --> (batch, seq, h, d_k) --> (batch, seq, d_model)
    x, self.attention_scores = MultiHeadAttentionBlock.attention(query, key,
        value, mask, self.dropout)
    x = x.transpose(1, 2).contiguous().view(x.shape[0], -1, self.h * self.d_k)

    return self.w_o(x) # Multiplica por la matriz de salida
```

La clase `MultiHeadAttentionBlock` aplica la atención de multiples cabezales, ya sea para el Encoder como para el Decoder, en cuyo caso agrega una máscara para evitar prestar atención a palabras que aparecen después en la traducción. En las lineas 37, 38 y 39 se obtienen los vectores de consulta, clave y valor para cada token en la secuencia haciendo el producto punto entre la entradas y las respectivas matrices (líneas 12, 13 y 14). Luego, en las lineas 43, 44 y 45 se dividen esos vectores en la cantidad de cabezales, así cada cabezal trabaja con una fracción de los vectores de cada token. La función `attention` aplica la fórmula de atención presentada en Vaswani et al. (2017) para calcular los puntajes de atención entre cada par de palabras, y luego los transforma en probabilidades usando softmax. Finalmente, se obtiene el vector de salida para cada token en la frase, haciendo una suma ponderada de los vectores de valor de cada token multiplicado por el puntaje correspondiente. Notar que en el caso de necesitar una máscara, simplemente se le asigna valor de atención mínimo a los token a los cuales no se les tiene que prestar atención por estar después de la palabra que se está procesando en la secuencia.

4.6. Encoder

A continuación presentamos el código de cada bloque Encoder, y el componente Encoder que encadena varios bloques de Encoder.

```
1  class EncoderBlock(nn.Module):
2
3    def __init__(self, features: int, self_attention_block:
         MultiHeadAttentionBlock, feed_forward_block: FeedForwardBlock, dropout:
         float) -> None:
4      super().__init__()
5      self.self_attention_block = self_attention_block
6      self.feed_forward_block = feed_forward_block
7      self.residual_connections = nn.ModuleList([ResidualConnection(features,
           dropout) for _ in range(2)])
8
9    def forward(self, x, src_mask):
10     x = self.residual_connections[0](x, lambda x: self.self_attention_block(x,
           x, x, src_mask))
11     x = self.residual_connections[1](x, self.feed_forward_block)
12     return x
```

Un bloque Encoder simplemente aplica la atención de multiples cabezales, la conexión residual, la red neuronal feed forward fully connected y luego otra conexión residual.

```
1  class Encoder(nn.Module):
2
3    def __init__(self, features: int, layers: nn.ModuleList) -> None:
4      super().__init__()
5      self.layers = layers
6      self.norm = LayerNormalization(features)
7
8    def forward(self, x, mask):
9      for layer in self.layers:
10       x = layer(x, mask)
11     return self.norm(x)
```

El Encoder simplemente pasa la entrada por cada uno de los bloques de Encoder.

4.7. Decoder

Ahora presentamos el código de cada bloque Decoder, y de manera similar al Encoder, el componente Decoder encadena varios bloques de Decoder.

```
class DecoderBlock(nn.Module):

  def __init__(self, features: int, self_attention_block:
      MultiHeadAttentionBlock, cross_attention_block: MultiHeadAttentionBlock,
      feed_forward_block: FeedForwardBlock, dropout: float) -> None:
    super().__init__()
    self.self_attention_block = self_attention_block
    self.cross_attention_block = cross_attention_block
    self.feed_forward_block = feed_forward_block
    self.residual_connections = nn.ModuleList([ResidualConnection(features,
        dropout) for _ in range(3)])

  def forward(self, x, Encoder_output, src_mask, tgt_mask):
    x = self.residual_connections[0](x, lambda x: self.self_attention_block(x,
        x, x, tgt_mask))
    x = self.residual_connections[1](x, lambda x: self.cross_attention_block(x,
        Encoder_output, Encoder_output, src_mask))
    x = self.residual_connections[2](x, self.feed_forward_block)
    return x
```

Un bloque Decoder simplemente aplica la atención de multiples cabezales sobre si mismo, la conexión residual, la atención de múltiples cabezales cruzada, residual, la red neuronal feed forward fully connected y luego otra conexión residual.

```
class Decoder(nn.Module):

  def __init__(self, features: int, layers: nn.ModuleList) -> None:
    super().__init__()
    self.layers = layers
    self.norm = LayerNormalization(features)

  def forward(self, x, Encoder_output, src_mask, tgt_mask):
    for layer in self.layers:
      x = layer(x, Encoder_output, src_mask, tgt_mask)
    return self.norm(x)
```

El Decoder simplemente pasa la entrada por cada uno de los bloques de Decoder.

4.8. Procesamiento de la salida

En esta sección se muestra el código del componente encargado de transformar el embedding de salida del bloque de Decoder en una asignación de probabilidades a cada token del vocabulario de salida.

```
class ProjectionLayer(nn.Module):

  def __init__(self, d_model, vocab_size) -> None:
    super().__init__()
    self.proj = nn.Linear(d_model, vocab_size)

  def forward(self, x) -> None:
    # (batch, seq, d_model) --> (batch, seq, vocab_size)
    return self.proj(x)
```

La capa de proyección hace una transformación lineal para pasar del vector de dimensión `d_model` para cada posición de la secuencia, a uno de dimensión `vocab_size`, que permitirá asignarle una probabilidad a cada token del vocabulario de salida.

4.9. El modelo Transformer

A continuación se presenta el código del modelo en sí, donde se combinan los distintos componentes para definir el modelo con arquitectura Transformer.

```
class Transformer(nn.Module):

  def __init__(self, Encoder: Encoder, Decoder: Decoder, src_embed:
      InputEmbeddings, tgt_embed: InputEmbeddings, src_pos: PositionalEncoding,
      tgt_pos: PositionalEncoding, projection_layer: ProjectionLayer) -> None:
    super().__init__()
    self.Encoder = Encoder
    self.Decoder = Decoder
    self.src_embed = src_embed
    self.tgt_embed = tgt_embed
    self.src_pos = src_pos
    self.tgt_pos = tgt_pos
    self.projection_layer = projection_layer

  def encode(self, src, src_mask):
    # (batch, seq, d_model)
    src = self.src_embed(src)
    src = self.src_pos(src)
    return self.Encoder(src, src_mask)

  def decode(self, Encoder_output: torch.Tensor, src_mask: torch.Tensor, tgt:
      torch.Tensor, tgt_mask: torch.Tensor):
    # (batch, seq, d_model)
    tgt = self.tgt_embed(tgt)
    tgt = self.tgt_pos(tgt)
    return self.Decoder(tgt, Encoder_output, src_mask, tgt_mask)

  def project(self, x):
    # (batch, seq, vocab_size)
    return self.projection_layer(x)
```

Finalmente el Transformer define las funciones para pasar una entrada por el Encoder, pasar por el Encoder, y obtener la proyección.

Capítulo 5

Resultados

En esta sección se presentan los resultados obtenidos a partir de una serie de experimentos realizados sobre el dominio Satellite. El objetivo principal de estos experimentos es evaluar el impacto de distintas configuraciones en el desempeño de la función *popBestOp* utilizando PUO como métrica de referencia.

Los experimentos fueron diseñados para analizar cómo factores críticos, como la calidad del modelo de traducción, la selección y composición del conjunto de testigos, y la definición de la función de distancia, afectan el resultado final de la función.

Antes de mostrar los experimentos y sus resultados, se presenta el dominio utilizado para entrenar el modelo de traducción y evaluar la función. Además, se explican las características generales comunes a todos los experimentos y las configuraciones que varían.

5.1. Dominios utilizados

Recordemos que el modelo de traducción se entrena para un dominio en particular, ya que cada dominio tiene sus estructuras propias que idealmente el modelo debería aprender.

Se describe a continuación el dominio *Satellite*, que fue presentado en la International Planning Competition (IPC) 2002 (Fox and Long, 2011). Se eligió este dominio porque es relativamente pequeño, con pocos esquemas de acciones y no demasiado grandes, lo que lo hace fácil de entender. Sin embargo, muchos planners tienen complicaciones con los problemas de Satellite, no es un dominio trivial. Es un dominio lo suficientemente simple como para poder entenderlo, pero es desafiante para los planners. Además es un dominio interesante en su aplicación, no es "de juguete". Por estos motivos se eligió Satellite para realizar los experimentos.

5.1.1. Satellite

Este dominio es un modelo simplificado del problema de programación de observación satelital. El problema completo implica la planificación y coordinación del uso de uno o más satélites para realizar observaciones, recopilar datos y transferirlos a una estación terrestre. Cada satélite está equipado con diversos instrumentos que presentan características específicas, como requisitos de calibración, producción de datos, consumo de energía y necesidades de calentamiento y enfriamiento.

Los satélites tienen la capacidad de apuntar a diferentes objetivos, pero esta capacidad puede estar restringida debido a limitaciones como oclusión de objetivos o sus capacidades de rotación. Los datos generados por los instrumentos deben almacenarse en el satélite hasta que se abra una ventana de oportunidad para su descarga, lo cual ocurre únicamente durante períodos predeterminados de comunicación con una estación terrestre.

Además, existen dificultades adicionales, como gestión del consumo de energía, uso de energía solar y mantenimiento de temperaturas operativas durante períodos de sombra. Sin embargo, muchos de estos desafíos fueron descartados ya que son difíciles de expresar en PDDL.

El objetivo del modelo consiste en encontrar la cobertura más eficiente de las observaciones, dadas las capacidades y restricciones de los satélites.

Este dominio cuenta con 5 esquemas de acciones y 13 predicados. A continuación, en el Cuadro 5.1.1 se detallan los esquemas de acciones que contiene este dominio, indicando para cada uno de ellos el tamaño de la interfaz del esquema (Int), la cantidad de predicados en el conjunto de precondiciones (Pre) y la cantidad de acciones en la lista de predicados a agregar (Add).

Esquema	**Int**	**Pre**	**Add**
TAKE_IMAGE	4	6	1
CALIBRATE	3	4	1
TURN_TO	3	2	2
SWITCH_ON	2	2	3
SWITCH_OFF	2	2	2

Cuadro 5.1: Esquemas de acción del dominio Satellite

En este dominio, contamos con 240 problemas de entrenamiento y 48 problemas de validación. Como se explicó en la Sección 3.2.2, para cada problema tenemos al menos un plan que lo resuelve, y hechos relajados asociados a algún plan relajado del problema, en ambas versiones secuenciales (en orden alfabético y en orden de aparición).

Para generar los datos de entrenamiento, seleccionamos una de las versiones ordenadas de los hechos relajados (por ejemplo, la secuencia ordenada por orden de aparición) y, para cada problema, emparejamos esta secuencia de hechos relajados con cada uno de los planes disponibles para el problema. Como resultado, en este dominio obtenemos un total de 2054 pares de entrenamiento.

A continuación, presentamos algunos de los factores principales que influyen en el desempeño de nuestra propuesta para la función $popBestOp$.

5.2. Temas críticos para un buen modelo

El desempeño de la solución propuesta para $popBestOp$ depende de al menos los siguientes tres aspectos fundamentales: la calidad del modelo de traducción, el método de generación del conjunto de testigos y la definición de la función de distancia. A continuación, se analiza cada uno de ellos:

- **El modelo de traducción**: el modelo debe ser capaz de aprender las relaciones entre los hechos y los planes. Uno de los factores que influye en el aprendizaje es

la calidad del conjunto de datos de entrenamiento. Factores como el tamaño del conjunto, la cantidad de ruido y la claridad de los patrones en los pares de datos afectan el desempeño del modelo. Un conjunto de datos pequeño, ruidoso o con ejemplos poco representativos puede dificultar que el modelo capture las relaciones necesarias.

- **Conjunto de testigos**: el modelo se usa para generar conjunto de testigos, que luego son usados para calcular la relevancia de cada acción. Existen distintas formas de obtener una traducción usando el modelo, aunque estas traducciones, por lo general, no serán un plan perfecto que resuelva el problema. El conjunto de testigos debe ser lo suficientemente robusto para manejar el ruido provocado por los errores del modelo. Para esto, es de utilidad considerar múltiples traducciones generadas mediante técnicas como greedy decoding y Beam Search. Al no restringirse a una única traducción potencialmente ruidosa, se puede mejorar la diversidad del conjunto de testigos. Sin embargo, incluir demasiadas traducciones puede introducir ruido en el conjunto de testigos, lo que podría afectar negativamente el desempeño.

- **Distancia final**: a cada acción se le asigna una distancia respecto al conjunto de testigos. Este puntaje, representa qué tan alejada está la acción de los hechos relajados del problema. Este puntaje se asigna basandose en el conjunto de testigos y los hechos relajados del problema. Si la función utilizada para calcular la diferencia es demasiado restrictiva podría penalizar operadores válidos que no coinciden exactamente con un testigo. Por otro lado, una función demasiado permisiva podría dejar pasar operadores irrelevantes como buenos. Encontrar un equilibrio adecuado es crucial para lograr un buen desempeño.

En el resto de este capítulo, realizaremos distintos experimentos para explorar distintos aspectos para cada uno de estos factores.

A continuación, se presenta la configuración general de los distintos experimentos, explicando cuáles serán los cambios entre ellos.

5.3. Configuración general

Los hiperparámetros del modelo, incluyendo la cantidad de bloques de Encoder y Decoder ($N_{Encoder}$ y $N_{Decoder}$), el tamaño del embedding d, y la cantidad de cabezales de atención h, fueron seleccionados utilizando el optimizador Optuna.

En los experimentos se evalúan distintas configuraciones con el objetivo de analizar el impacto de diferentes factores en el desempeño de la función $popBestOp$. Las configuraciones varían en términos de:

- **Datos de entrada del modelo:** se utilizan distintas variantes de los hechos relajados como representación inicial.

- **Forma de generar los testigos:** se consideran enfoques como Greedy Decode y Beam Search para generar las traducciones usando el modelo. De estas traducciones se obtienen los testigos.

- **Criterio de distancia final:** se evalúan métricas como la distancia mínima, la distancia promedio y el porcentaje de objetos no presentes en los hechos relajados.

Estas configuraciones permiten explorar cómo diferentes ajustes afectan la calidad de las soluciones generadas. En los experimentos presentados a continuación, se utiliza PUO como métrica para evaluar el desempeño final de la función *popBestOp* aplicada a un conjunto de 48 problemas de validación del dominio Satellite.

5.4. Recursos de cómputo y tiempos de ejecución

El modelo Transformer usado por la función *popBestOp* fue entrenado en la computadora `atom` (FaMAF-UNC), que cuenta con una CPU AMD EPYC 7643 de 48 núcleos y 96 hilos, 125 GiB de RAM y una GPU NVIDIA TITAN Xp con 12 GiB de VRAM. El entrenamiento de un modelo de traducción durante 50 épocas en esta máquina tardaba entre 6 y 7 horas.

En esta misma computadora, se ejecutaron los experimentos que serán presentados en esta sección. Aplicar la función *popBestOp* con una configuración específica sobre los 48 problemas de validación y medir el PUO resultante, tomó aproximadamente 40 minutos.

Además, se usó la computadora `nabucodonosor` (FaMAF-UNC) para realizar la optimización de hiperparámetros usando Optuna. Esta computadora cuentacon una CPU Intel Xeon E5-2680 v2 de 20 núcleos y 40 hilos, 62 GiB de RAM, y dos tarjetas gráficas: una NVIDIA A30 con 24 GiB de VRAM y una NVIDIA A10 con 23 GiB de VRAM. En esta máquina, tardaba aproximadamente 4 días correr Optuna para comparar a lo sumo 50 configuraciones distintas de hiperparámetros para el modelo de traducción, variando tasa de aprendizaje, cantidad de cabezales, cantidad de bloques de encoder, cantidad de bloques de decoder y tamaño de embedding.

5.5. Experimento #1: El impacto de testigos ruidosos

En este experimento se analiza cómo la calidad del conjunto de testigos influye en el desempeño final de la función *popBestOp*. Inicialmente, se utilizan testigos ideales, formados exclusivamente por los Buenos Operadores de cada problema (es decir, no se genera este conjunto mediante un modelo de traducción, sino que se seleccionan manualmente). Esto sería una especie de "Gold Standard", en donde el modelo es un oráculo perfecto, obteniendo el plan exacto a partir de los hechos relajados. Claramente, esto no es una asunción realista. Luego, se introduce ruido al conjunto de testigos reemplazando progresivamente algunos Buenos Operadores por Malos Operadores.

Recordemos que vamos a trabajar con subconjuntos de todos los posibles Buenos y Malos Operadores. El subconjunto de Buenos Operadores está conformado por todos los operadores que aparecen en un determinado plan que resuelve el problema. Para el subconjunto de Malos Operadores, se seleccionan hasta 50000 operadores del conjunto original, intentando que haya diversidad entre ellos.

Cuando el conjunto de testigos está compuesto únicamente por el subconjunto de Buenos Operadores, cada uno de ellos tiene una distancia individual de 0 consigo mismo como testigo, mientras que los Malos Operadores presentan distancias mayores. Si la

distancia final se define como la menor distancia con algún testigo, todos los Buenos Operadores con los que estamos trabajando tendrán una distancia final de 0, mientras que los Malos Operadores tendrán distancias mayores. Como resultado, los Buenos Operadores recibirán probabilidades más altas que los Malos Operadores.

Dado que nuestros Buenos Operadores corresponden a todos los operadores que aparecen en un plan determinado, instanciar todos los operadores de algún plan requiere seleccionar aquellos cuya probabilidad sea mayor o igual a la probabilidad asignada al Buen Operador con menor probabilidad. Como todos los Malos Operadores tienen una probabilidad menor a la de los Buenos Operadores, en este escenario ideal solo se instancian los Buenos Operadores, lo que resulta en un PUO cercano a 1.

Recordemos que el PUO se define como la cantidad de operadores no instanciados sobre el total de operadores considerados (es decir, la unión de los subconjuntos de Buenos y Malos Operadores utilizados). En promedio, los planes de los problemas de validación contienen 35 Buenos Operadores, mientras que el conjunto de Malos Operadores tiene un tamaño promedio de 9860. Por lo tanto, en la configuración ideal, donde solo se instancian los Buenos Operadores, el PUO promedio sería de $\frac{9860}{9895} \approx 1$.

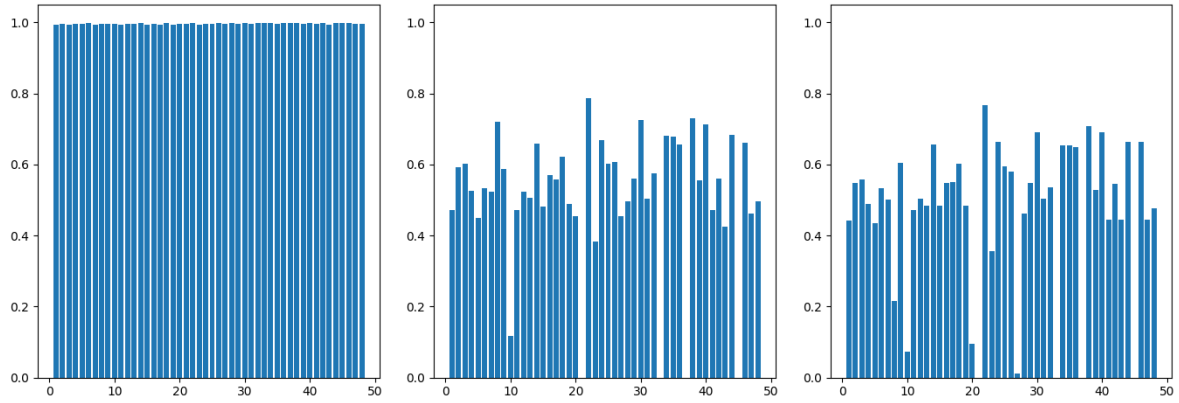
Sin embargo, al introducir ruido en el conjunto de testigos reemplazando Buenos Operadores por Malos Operadores, los Buenos Operadores descartados adquieren distancias mayores a 0. Simultáneamente, los Malos Operadores añadidos al conjunto de testigos obtienen una distancia de 0 consigo mismos. Como resultado, algunos Buenos Operadores pasan a tener una distancia mayor que algunos Malos Operadores, lo que a su vez provoca que ciertos Malos Operadores reciban una probabilidad mayor que algunos Buenos Operadores. En este escenario, para garantizar la selección de todos los Buenos Operadores, también se instancian algunos Malos Operadores, lo que reduce el PUO. A medida que aumenta el nivel de ruido (es decir, la proporción de Buenos Operadores reemplazados por Malos Operadores), el PUO se deteriora progresivamente.

Las distintas configuraciones probadas en este experimento se presentan en el Cuadro 5.2.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	No se usa	Buenos Operadores	Distancia mínima
Conf. 2	Hechos relajados en orden alfabético	No se usa	Buenos Operadores, reemplazado uno por un Mal Operador	Distancia mínima
Conf. 3	Hechos relajados en orden alfabético	No se usa	Buenos Operadores, reemplazando dos por Malos Operadores	Distancia mínima

Cuadro 5.2: Configuraciones para el Experimento #1

La Figura 5.1 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.1: Resultados Experimento #1
Número de problema sobre el eje x y PUO obtenido sobre el eje y

En la Subfigura 5.1a vemos que los testigos perfectos garantizan PUO de casi 1 para cada problema de validación, como se esperaba, ya que todos los Operadores Buenos tienen una distancia 0 consigo mismos, asegurando que únicamente estos sean instanciados.

En la Subfigura 5.1b, se observa que al introducir un testigo incorrecto, 16 de los 48 problemas de validación presentan un PUO menor a 0,5, lo que indica que en estos problemas se instancia al menos la mitad de los Malos Operadores. En cuatro de estos problemas, el PUO es 0, es decir, todos los Malos Operadores son instanciados. Este comportamiento ocurre principalmente en casos donde el Buen Operador eliminado era el único representante de su tipo. La ausencia de un testigo de ese tipo provoca que el operador correspondiente tenga una probabilidad de 0, obligando a instanciar todos los operadores hasta cubrir este caso. Este resultado resalta la importancia de contar con al menos un testigo para cada tipo de operador, ya que su ausencia puede llevar a un PUO de 0 y un incremento significativo en los Malos Operadores instanciados.

En la Subfigura 5.1c se muestra que con dos testigos incorrectos, 21 de 48 problemas de validación presentan PUO menor a 0,5. Por ejemplo, el problema 8, que en la Subfigura 5.1b tenía un PUO cercano a 0,7, ahora cae a aproximadamente 0,2.

El experimento muestra que la calidad del conjunto de testigos tiene un impacto crítico en el desempeño del modelo. Mientras que los testigos perfectos permiten alcanzar un PUO ideal de 1, incluso pequeñas cantidades de ruido, como reemplazar un único Operador Bueno por un Operador Malo, deterioran notablemente los resultados. Este impacto negativo aumenta a medida que se introduce mayor ruido en el conjunto de testigos.

5.6. Experimento #2: Distintas formas de generar testigos

Habiendo visto el impacto de un buen conjunto de testigos en el desempeño final de la función $popBestOp$, ahora comparamos distintas formas de obtenerlos. Dada una traducción, usamos las acciones presentes en la misma como testigos, pero hay diversas formas de generar una traducción. Consideramos dos: Beam Search y Greedy Decode.

Como se explicó previamente, Greedy Decode es una estrategia que, en cada paso,

elige el token más probable para continuar la secuencia. Es un método simple y rápido, pero su mayor limitación es que no explora secuencias alternativas, perdiendo la oportunidad de obtener una mejor traducción. En contraste, Beam Search mantiene múltiples secuencias parciales a la vez. En lugar de elegir solo el token más probable en cada paso, Beam Search mantiene las n mejores secuencias según su probabilidad acumulada, generando así n posibles traducciones. El parámetro n, llamado tamaño del beam, define cuántas secuencias parciales se mantienen en cada paso.

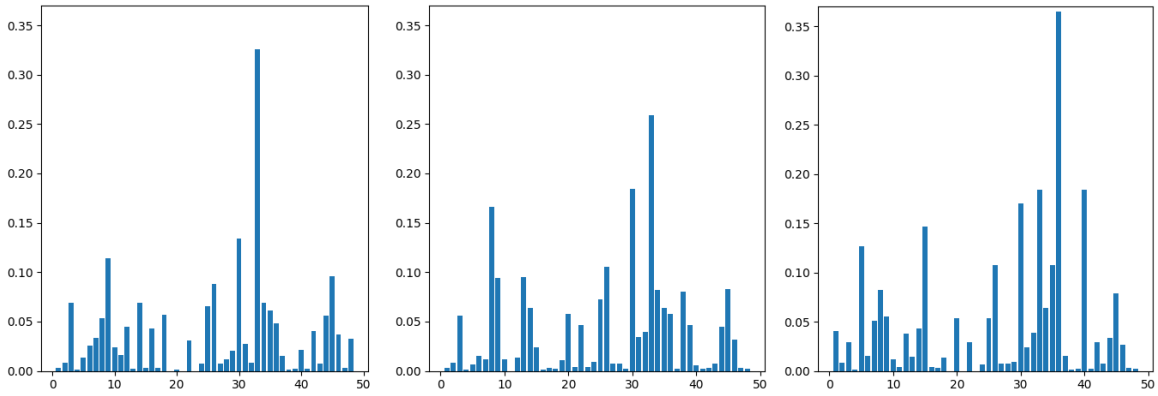
El uso de más traducciones puede generar más testigos parecidos a los Operadores Buenos, haciendo que se les asigne a estos últimos una probabilidad mayor de aparecer en el plan final. Sin embargo, esto también puede introducir más ruido en los testigos, con lo cual podría asignarse probabilidad más alta a los Operadores Malos también.

En el Cuadro 5.3 se muestran las distintas configuraciones usadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode	Distancia mínima
Conf. 2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Beam Search (tamaño 4)	Distancia mínima
Conf. 3	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia mínima

Cuadro 5.3: Configuraciones para el Experimento #2

La Figura 5.2 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.2: Resultados Experimento #2
Número de problema sobre el eje x y PUO obtenido sobre el eje y

En la Subfigura 5.2a, se observa que sólo 3 de los 48 problemas de validación superan un PUO de 0,1 al generar las traducciones utilizando exclusivamente Greedy Decode. Esto muestra la limitación de esta estrategia para producir testigos suficientemente diversos.

En la Subfigura 5.2b, se muestra que el uso de Beam Search con un tamaño de 4 permite una ligera mejora respecto a la estrategia anterior, logrando que 4 de los 48 problemas de validación superen un PUO de 0,1. Esto muestra que generar múltiples traducciones a partir de las cuales se construyen los testigos puede mejorar el rendimiento del modelo.

Finalmente, en la Subfigura 5.2c, vemos que 8 de los 48 problemas de validación alcanzan un PUO mayor a 0,1 al usar Beam Search y Greedy Decode. Esta estrategia es la más efectiva de las tres probadas, mostrando nuevamente que incrementar la diversidad en los testigos mediante el uso de múltiples traducciones contribuye a mejorar los resultados.

En general, vemos que incrementar la diversidad en los testigos generados mejora el desempeño del modelo en términos de PUO, sin introducir un nivel significativo de ruido.

5.7. Experimento #3: Impacto de la función de distancia

Recordemos que una vez obtenido el conjunto de testigos, se calcula la distancia entre cada operador y el conjunto de testigos. La distancia entre un operador y un testigo del mismo tipo está dado por la cantidad de tokens distintos entre el operador y el testigo, esto luego se divide por la cantidad de tokens del operador. Ahora, para la distancia final entre el operador y todo el conjunto de testigos, probamos dos estrategias distintas:

- **Distancia mínima:** consiste en calcular la distancia entre el operador y cada uno de los testigos del mismo tipo, y tomar la menor de ellas como la distancia final asignada al operador.

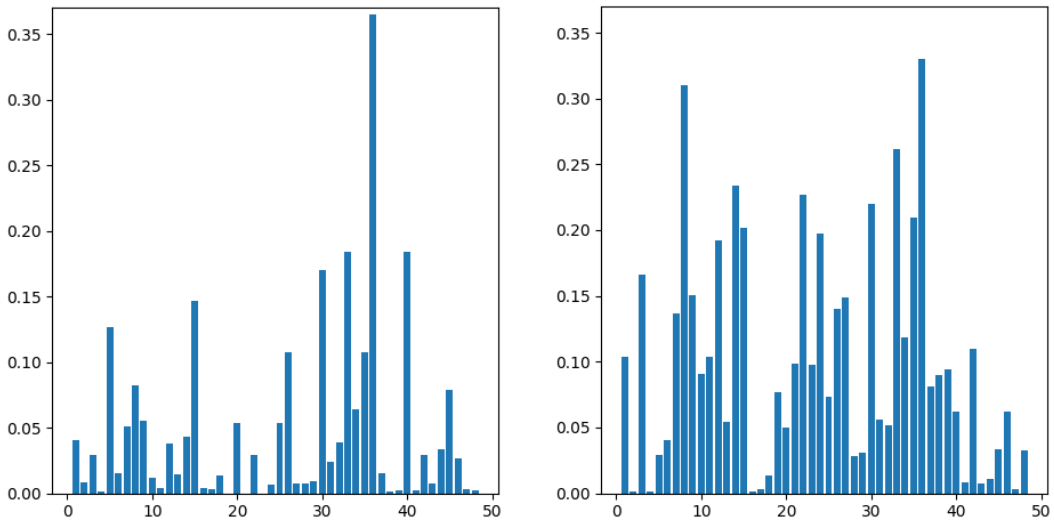
- **Distancia promedio:** consiste en calcular la distancia entre el operador y cada uno de los testigos del mismo tipo, sumar dichas distancias y dividir el resultado por la cantidad total de testigos del mismo tipo. Esta estrategia incorpora la frecuencia de aparición de los testigos en las traducciones generadas. Por ejemplo, si un testigo aparece en más de una traducción (lo que puede interpretarse como mayor confianza del modelo en ese operador), esto será reflejado en la distancia final.

En el Cuadro 5.4 se muestran las distintas configuraciones usadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia mínima
Conf. 2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia promedio

Cuadro 5.4: Configuraciones para el Experimento #3

La Figura 5.3 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2

Figura 5.3: Resultados Experimento #3
Número de problema sobre el eje *x* y PUO obtenido sobre el eje *y*

En la Subfigura 5.3a, se presentan los resultados obtenidos utilizando la estrategia de distancia mínima. En este caso, solo 8 de 48 problemas de validación superan un PUO de 0,1. Esto puede deberse a que si una traducción contiene un operador ruidoso que es igual a un Mal Operador, este Mal Operador tendrá una distancia final de 0, y otros Malos Operadores que puedan ser parecidos, también tendrán una distancia baja. Estos Malos Operadores tendrán entonces una probabilidad alta asignada, y podrían tener probabilidad mayor al Buen Operador con menor probabilidad, disminuyendo así el PUO.

En la Subfigura 5.3b, se muestran los resultados al utilizar la estrategia de distancia promedio. Ahora, 19 de los 48 problemas de validación tienen PUO mayor a 0,1. Esta configuración incorpora la información de todos los testigos generados para un operador, disminuyendo el impacto de valores ruidosos. Como resultado, el PUO mejora significativamente en varios problemas, reflejando una mayor robustez frente al ruido.

En términos generales, los resultados muestran que la estrategia de distancia promedio es más efectiva que la de distancia mínima. Esto puede deberse a que la primera considera la frecuencia de aparición de los testigos, penalizando aquellos generados por traducciones ruidosas.

5.8. Experimento #4: Distinto tamaño para Beam Search

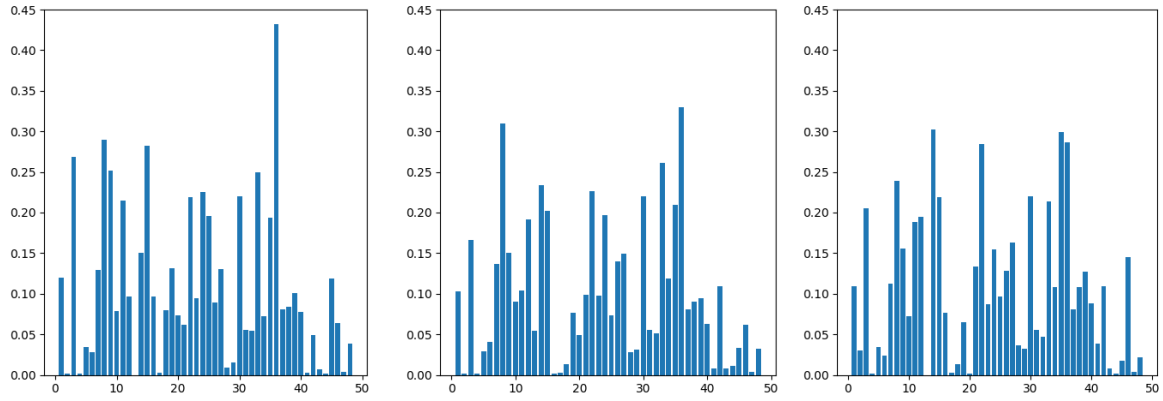
En el experimento anterior, se observó que utilizar la distancia promedio permite mitigar el impacto de los testigos ruidosos en el desempeño final del modelo. Esto motiva explorar cómo afecta el desempeño final al variar el tamaño del beam en el proceso de Beam Search. En este experimento, se analiza el uso de diferentes tamaños de beam, incluyendo un valor menor al utilizado previamente.

En el Cuadro 5.5 se presentan las distintas configuraciones utilizadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 2)	Distancia promedio
Conf. 2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia promedio
Conf. 3	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio

Cuadro 5.5: Configuraciones para el Experimento #4

La Figura 5.4 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.4: Resultados Experimento #4
Número de problema sobre el eje x y PUO obtenido sobre el eje y

En la Subfigura 5.4a, se observa que 19 de los 48 problemas de validación superan un PUO de 0,1 al usar un tamaño de beam de 2. Luego, 10 problemas superan un PUO de 0,2, y tan solo un problema supera PUO de 0,3.

En la Subfigura 5.4b, se muestra que con un tamaño de beam de 4, nuevamente 19 problemas de validación alcanzan un PUO mayor a 0,1. Además, 8 superan un PUO de 0,2 y dos problemas alcanzan PUO de 0,3.

Finalmente, en la Subfigura 5.4c, se observa que al usar tamaño de beam de 7, 23 problemas de validación superan un PUO de 0,1. Luego, 9 superan un PUO de 0,2, y 1 superan un PUO de 0,3.

Vemos que usando un beam de 7 se obtiene un rendimiento ligeramente mejor (es el que tiene la mayor cantidad de problemas con puo de al menos 0,1). La diversidad de traducciones, emparejado con el uso de la distancia promedio que permite reducir el impacto de los testigos ruidosos en el puntaje final asignado a los operadores, parece generar los mejores resultados.

Se quiso probar con un tamaño de beam mayor a 7, pero se debe recordar que un tamaño de beam de n implica calcular y almacenar n traducciones en simultáneo, lo que aumenta el consumo de memoria. Los experimentos se realizaron en una máquina con una GPU de 12 GiB de VRAM, y al intentar usar un tamaño de beam de 8, se generó un error de memoria.

5.9. Experimento #5: Uso de porcentaje de objetos no presentes en los hechos relajados

Una forma de medir qué tan diferente es una acción respecto a los hechos relajados es calcular el porcentaje de objetos de la acción que no están presentes en estos hechos. Esto se debe a que, generalmente, las precondiciones y postcondiciones de una acción están asociadas a los objetos involucrados en ella. Si una acción forma parte del plan, entonces sus precondiciones y postcondiciones deberían estar reflejadas en los hechos relajados. Por lo tanto, si los objetos de una acción no aparecen en los hechos relajados de un problema, es razonable suponer que dicha acción no será parte del plan asociado a esos hechos relajados.

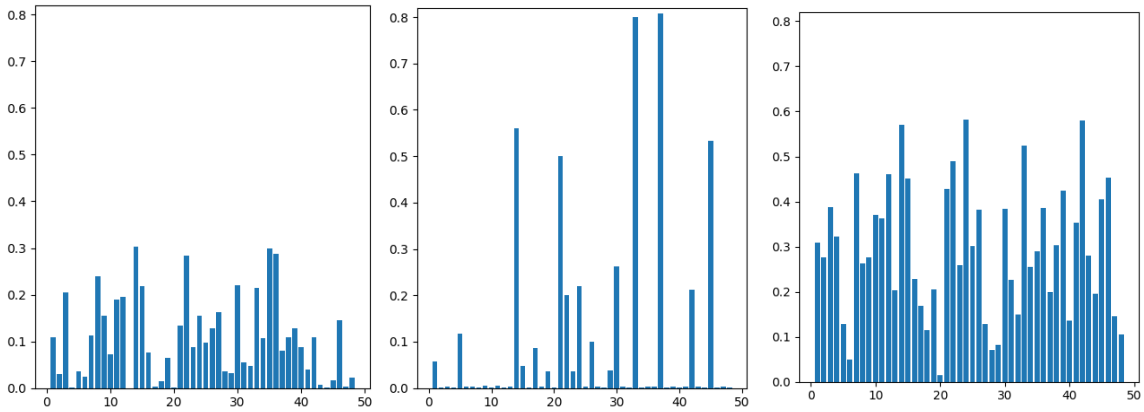
En este experimento, se analiza el impacto de usar el porcentaje de objetos que no están presentes en los hechos relajados para calcular la distancia final de cada operador. Además, se evalúa la efectividad de combinar esta métrica con la distancia promedio.

En el Cuadro 5.6 se muestran las distintas configuraciones usadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio
Conf. 2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Porcentaje de objetos no presentes en los hechos relajados
Conf. 3	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados

Cuadro 5.6: Configuraciones para el Experimento #5

La Figura 5.5 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.5: Resultados Experimento #5
Número de problema sobre el eje *x* y PUO obtenido sobre el eje *y*

En la Subfigura 5.5a, se observa que 23 de los 48 problemas de validación superan un PUO de 0,1, y solo 1 problema de validación supera un PUO de 0,3. Al usar solamente la distancia promedio con el conjunto de testigos para calcular la distancia final de cada operador. Con esta estrategia, no hacemos ningún tipo de "limpieza" sobre las traducciones generadas, podrían tener acciones que no podrían aparecer en el plan ya que sus objetos no están en los hechos relajados, pero no las descartamos.

En la Subfigura 5.5b, se muestra que en este caso solo 11 de los 48 problemas de validación presentan PUO mayor a 0,1, y solo 5 problemas de validación superan un PUO de 0,3. Al usar solamente el porcentaje de objetos no presentes en los hechos relajados para calcular la distancia final de cada operador. Esto muestra que no es suficiente con considerar la información que proveen directamente los hechos relajados, un Mal Operador puede tener objetos que aparecen en los hechos relajados, y no por eso significa que será necesario en el plan. Sin embargo, en este escenario al Mal Operador se le asignaría una distancia final de 0, lo que luego se convierte en una probabilidad alta.

Finalmente, en la Subfigura 5.5c, se observa que 44 problemas de validación alcanzan un PUO mayor a 0,1 al usar la distancia promedio y el porcentaje de objetos no presentes en los hechos relajados para calcular la distancia final de cada operador. Más aun, casi la mitad de los problemas de validación (23 de 48) presentan PUO mayor a 0,3. Vemos que al combinar la información brindada por los testigos y los hechos relajados, se obtienen los mejores resultados.

En general, vemos que el uso de ambas estrategias permite aprovechar los patrones aprendidos por el modelo de traducción (a partir del cuál se obtienen las traducciones de hechos relajados en planes, y luego de ahi se obtienen los testigos) pero además permite considerar qué tan "factible" es que un operador esté en el plan dados los objetos presentes en los hechos relajados.

5.10. Experimento #6: Distintos datos de entrada

Este experimento evalúa el impacto de los datos de entrada utilizados. Se explicó que dado un problema y un plan que lo resuelve, los hechos relajados son los hechos que

son verdaderos luego de aplicar una versión relajada del plan a partir del estado inicial del problema. Para pasar el conjunto de hechos relajados a una secuencia consideramos dos posibilidades: la primera es ordenar el conjunto alfabéticamente, y la segunda es ordenarlos según el momento de aparición (primero los hechos agregados por la primera acción del plan, luego los agregados por la segunda acción del plan, y así sucesivamente).

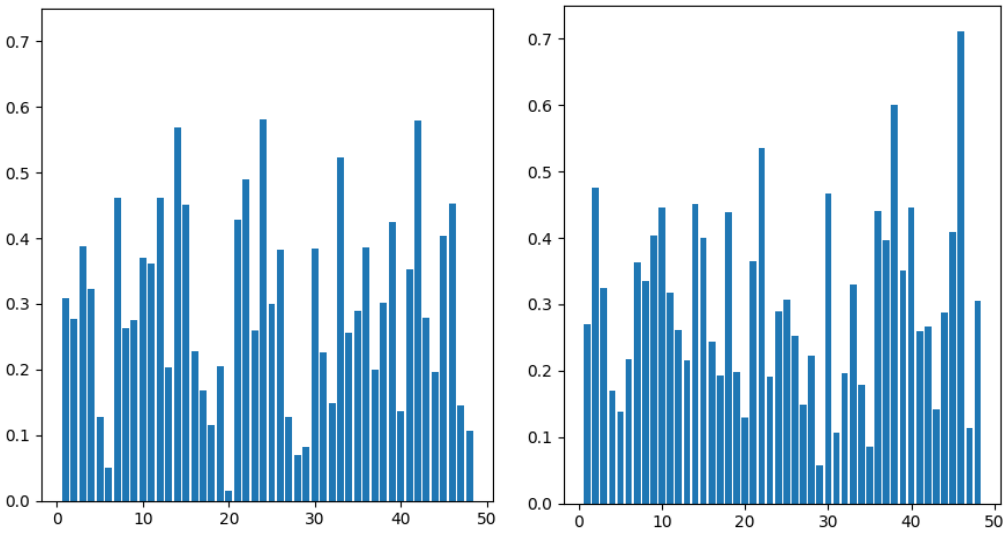
Ahora, queremos comparar el desempeño utilizando cada una de estas formas de ordenar el conjunto de hechos relajados. Para esto, entrenamos dos modelos distintos: uno de ellos con los hechos relajados en orden alfabético, y otro con los hechos relajados en orden de aparición. Los hiperparámetros de cada uno fueron encontrados con Optuna.

En el Cuadro 5.7 se muestran las distintas configuraciones usadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados
Conf. 2	Hechos relajados en orden de aparición	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados

Cuadro 5.7: Configuraciones para el Experimento #6

La Figura 5.6 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1

(b) PUO con Config. 2

Figura 5.6: Resultados Experimento #6
Número de problema sobre el eje x y PUO obtenido sobre el eje y

En la Subfigura 5.6a, se observa que usando los hechos relajados en orden alfabético 44 de 48 problemas de validación presentan un PUO mayor a 0,1, 35 tienen PUO mayor a 0,2 y 23 problemas tienen PUO mayor a 0,3. El PUO promedio entre los 48 problemas de validación es de 0,294.

Por otro lado, en la Subfigura 5.6b se muestra que usando los hechos relajados en orden de aparición 46 de 48 problemas de validación presentan un PUO mayor a 0,1, 34 tienen PUO mayor a 0,2 y 23 problemas tienen PUO mayor a 0,3. El PUO promedio entre los 48 problemas de validación es de 0,301.

En general, ambas configuraciones muestran resultados similares, con un rendimiento ligeramente superior usando el orden de aparición. Esto puede deberse a que el orden de aparición contienen más información que el orden alfabético, refleja de manera explícita cómo se van introduciendo los hechos en el estado del problema a medida que se aplican las acciones del plan. Esto podría permitirle al modelo identificar patrones entre los hechos y las acciones que los generan. El orden de aparición tiene información temporal, que el modelo parece poder aprovechar.

5.11. Experimento #7: Limpiar el conjunto de entrenamiento

En este experimento, evaluamos el impacto de limpiar el conjunto de datos de entrenamiento al eliminar pares de entrenamiento considerados ruidosos. Como se explicó anteriormente, el conjunto de hechos relajados de un problema está asociado con un plan específico. Sin embargo, al construir los pares de entrenamiento, asociamos un único conjunto de hechos relajados con todos los planes que resuelven el problema. Esto amplía el conjunto de datos de entrenamiento, pero puede introducir ruido, ya que los hechos relajados pueden no corresponder al plan con el que se emparejan.

Por ejemplo, si tenemos un plan que usa *avion*1 y otro que usa *avion*2, pero asociamos los hechos relajados del primer plan con el segundo, el modelo puede tener dificultades para establecer la relación entre los objetos del plan y los hechos relajados. Sin embargo, incluso estos pares "ruidosos" pueden ser útiles para que el modelo aprenda aspectos de la gramática del dominio.

La métrica utilizada para medir el ruido de un par de entrenamiento es el porcentaje de objetos presentes en el plan que no aparecen en los hechos relajados, dividido por el total de objetos en el plan. Un valor de ruido igual a 1 significa que ningún objeto del plan está presente en los hechos relajados, lo que hace que el par sea altamente ruidoso, ya que el modelo no tiene información sobre esos objetos en los hechos relajados.

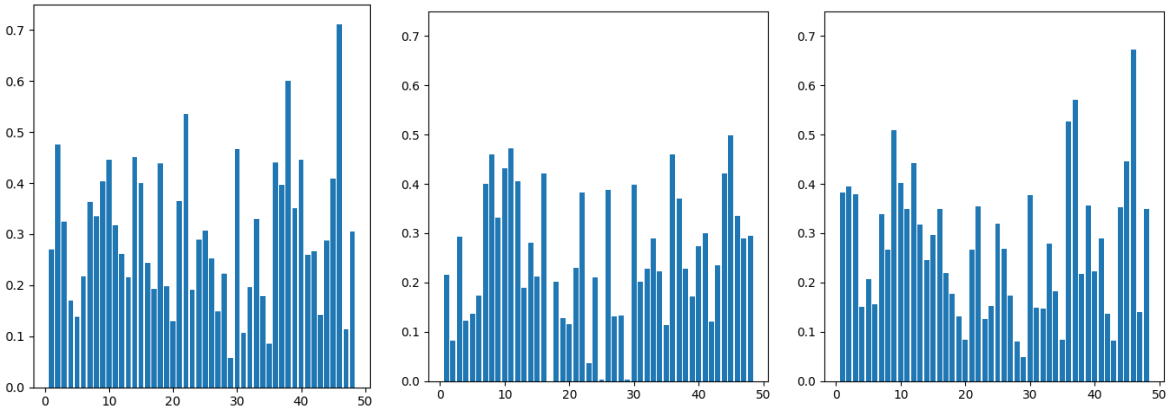
Entonces, el objetivo de este experimento es analizar cómo afecta al rendimiento del modelo descartar pares de entrenamiento con niveles de ruido superiores a ciertos umbrales.

En el Cuadro 5.8 se presentan las distintas configuraciones utilizadas en este experimento.

	Datos de Entrada	Configuración del Modelo	Generación de testigos	Distancia Final
Conf. 1	Hechos relajados en orden de aparición con todos los pares (2054 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados
Conf. 2	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,5 (1961 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados
Conf. 3	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,25 (1243 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados

Cuadro 5.8: Configuraciones para el Experimento #7

La Figura 5.7 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.7: Resultados Experimento #7
Número de problema sobre el eje x y PUO obtenido sobre el eje y

En la Subfigura 5.7a, donde se usan todos los pares de datos de entrenamiento, se observa que 46 de los 48 problemas de validación tienen un PUO mayor a 0,1, 34 problemas tienen PUO mayor a 0,2, y 23 problemas superan 0,3.

En la Subfigura 5.7b, donde se descartaron pares con ruido mayor a 0,5, 43 problemas de validación tienen PUO mayor a 0,1, 32 superan 0,2, y 15 superan 0,3.

Finalmente, en la Subfigura 5.7c, vemos que al descartar pares con ruido mayor a 0,25, 43 problemas de validación presentan PUO mayor a 0,1, 30 superan 0,2, y 20 superan 0,3.

Se observa que usar todos los datos, incluso los ruidosos, muestra el mejor desempeño general en este experimento. Esto puede deberse a que, aunque los pares ruidosos pue-

den dificultar la relación entre hechos relajados y planes, estos siguen siendo útiles para aprender la gramática del dominio.

Sin embargo, también podría suceder que al correr Optuna con el conjunto de entrenamiento "limpio", se encuentren nuevos hiperparámetros óptimos que permitan mejorar las métricas.

5.12. Experimento #8: Ajuste de hiperparámetros sobre conjunto de entrenamiento limpio

En el experimento anterior vimos que al entrenar el modelo con una menor cantidad de datos (pero con menor cantidad de ruido), el desmepeño empeoró. Esto puede deberse a que mayor cantidad de datos, por más de que sean ruidosas, sirven como ejemplos de entrenamiento, o que los hiperparámetros encontrados para el conjunto de entrenamiento total, no son los óptimos para el conjunto de entrenamiento con menos ruido. En este experimento entonces vamos a evaluar el desempeño del modelo entrenado con ruido menor al 0,5, pero habiendo corrido Optuna para encontrar los hiperparámetros óptimos para este conjunto de entrenamiento.

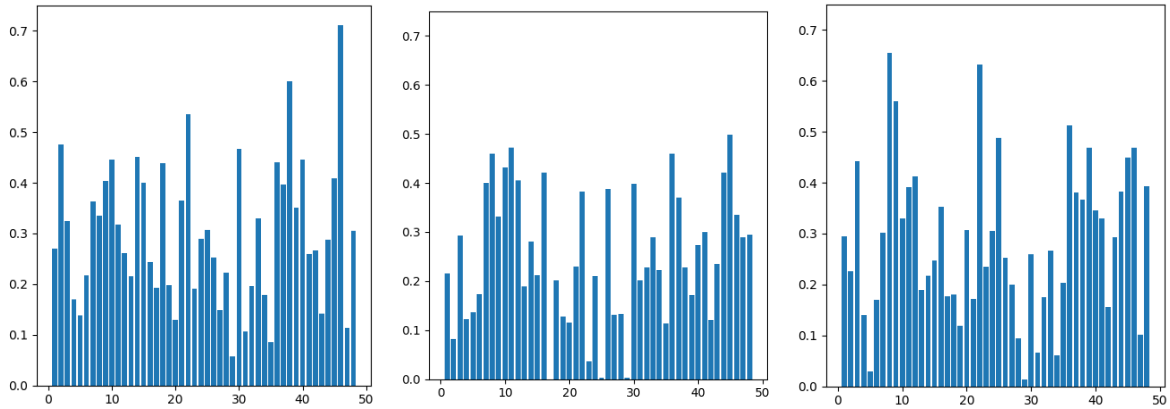
Vamos a comparar los resultados obtenidos con el desempeño sin optimizar, y con el desempeño con el conjunto de entrenamiento completo.

En el Cuadro 5.9 se presentan las distintas configuraciones utilizadas en este experimento.

	Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Conf. 1	Hechos relajados en orden de aparición con todos los pares (2054 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados
Conf. 2	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,5 (1961 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados
Conf. 3	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,5 (1961 pares de entrenamiento)	$N_{Encoder}$: 6 $N_{Decoder}$: 2 d: 64 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados

Cuadro 5.9: Configuraciones para el Experimento #8

La Figura 5.8 presenta los resultados obtenidos para distintas configuraciones evaluadas sobre el conjunto de validación.



(a) PUO con Config. 1 (b) PUO con Config. 2 (c) PUO con Config. 3

Figura 5.8: Resultados Experimento #8
Número de problema sobre el eje *x* y PUO obtenido sobre el eje *y*

En la Subfigura 5.8a, donde se usan todos los pares de datos de entrenamiento, se observa que 46 de los 48 problemas de validación tienen un PUO mayor a 0,1, 34 problemas tienen PUO mayor a 0,2, y 23 problemas superan 0,3.

En la Subfigura 5.8b, donde se descartaron pares con ruido mayor a 0,5, 43 problemas de validación tienen PUO mayor a 0,1, 32 superan 0,2, y 15 superan 0,3.

Finalmente, en la Subfigura 5.8c, vemos que al descartar pares con ruido mayor a 0,25, 43 problemas de validación presentan PUO mayor a 0,1, 33 superan 0,2, y 22 superan 0,3.

En este experimento observamos que al ajustar los hiperparámetros al conjunto de entrenamiento limpio se obtienen resultados ligeramente mejores que al entrenarlo con los hiperparámetros optimizados para el conjunto de entrenamiento completo. Sin embargo, sigue siendo mejor usar todos los datos del conjunto de entrenamiento, ruidosos o no. Sería interesante entonces construir un conjunto de entrenamiento más grande, ya que parecería que por más de que sean ruidosos el modelo necesita esos datos para poder aprender.

5.13. PUO promedio sobre validación de todos los experimentos

A continuación, en el Cuadro 5.10 se presenta un resumen de los experimentos realizados, mostrando el PUO promedio sobre los 48 problemas de validación para todas las configuraciones presentadas en los distintos experimentos en este capítulo.

Cuadro 5.10: PUO promedio sobre validación para cada configuración

Exp.	Datos de Entrada	Configuración del Modelo	Generación de testigos	Distancia Final	PUO promedio
1.1	Hechos relajados en orden alfabético	No se usa	Buenos Operadores	Distancia mínima	0,996
1.2	Hechos relajados en orden alfabético	No se usa	Buenos Operadores, reemplazado uno por un Mal Operador	Distancia mínima	0,512
1.3	Hechos relajados en orden alfabético	No se usa	Buenos Operadores, reemplazando dos por Malos Operadores	Distancia mínima	0,470
2.1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode	Distancia mínima	0,038
2.2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Beam Search (tamaño 4)	Distancia mínima	0,040
2.3 3.1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia mínima	0,048
3.2 4.2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 4)	Distancia promedio	0,101
4.1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 2)	Distancia promedio	0,110
4.3 5.1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio	0,107
5.2	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Porcentaje de objetos no presentes en los hechos relajados	0,098

Continúa en la siguiente página

Exp.	Datos de Entrada	Configuración del Modelo	Generación de testigos	Distancia Final	PUO promedio
5.3 6.1	Hechos relajados en orden alfabético	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados	0,294
6.2 7.1 8.1	Hechos relajados en orden de aparición	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados	0,301
7.2 8.2	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,5 (1961 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados	0,251
7.3	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,25 (1243 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados	0,274
8.3	Hechos relajados en orden de aparición descartando pares con ruido mayor a 0,5 (1961 pares de entrenamiento)	$N_{Encoder}$: 6 $N_{Decoder}$: 2 d: 64 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados	0,288

Capítulo 6

Conclusión y Trabajos Futuros

6.1. Conclusiones generales

En este trabajo se exploró el uso de un modelo de traducción para transformar hechos relajados de un problema en planes del mismo, y a partir de estos planes, determinar la relevancia de cada operador para resolver dicho problema. La relevancia asignada a cada operador se utiliza para guiar el proceso de grounding de un plan, instanciando únicamente los operadores con mayor relevancia. Esta heurística tiene como objetivo reducir la cantidad de operadores instanciados en comparación con el grounding clásico, al descartar aquellos operadores que son alcanzables de forma relajada pero que no serán necesarios para resolver el problema.

Con esta estrategia, el planner trabajará con un conjunto reducido de operadores, pero idealmente incluirá todos los necesarios para encontrar una solución. Al reducir el número de operadores, el planner podría encontrar un plan más rápidamente (si es que existe), ya que hay menos operadores que considerar al momento de generar el plan. Además, en ciertos problemas, el grounding clásico puede resultar en una cantidad tan grande de operadores instanciados que el sistema no tiene suficiente memoria para almacenarlos, impidiendo que el planner pueda ejecutarse. Con esta reducción, esos problemas podrían ser llegar a ser procesados por el planner.

Es importante señalar que la estrategia propuesta para la función $popBestOp$ no es perfecta, en el sentido de que no podemos garantizar que los operadores a los que le asigna baja probabilidad sean realmente irrelevantes, ni que aquellos con alta probabilidad sean necesarios. Esto se debe a que estamos usando un modelo de aprendizaje automático y estos modelos no son perfectos: no se basan en reglas predefinidas, sino que aprenden patrones en los ejemplos de entrenamiento y no siempre son capaces de generalizar correctamente a todos los casos posibles. Esta falta de certeza contrasta con el grounding clásico, donde es demostrable que si existe un plan para el problema, todas las acciones que aparecen en el mismo serán instanciadas.

Un aspecto interesante es que el modelo de traducción fue entrenado para generar un plan que resuelve un problema a partir de los hechos relajados del mismo, es decir está realizando una forma de searching. Sin embargo, a diferencia de los planners, cuyo proceso de búsqueda es claro y sigue reglas fijas, al usar el modelo de traducción no siempre se pueden dar explicaciones claras de por qué se toman determinadas decisiones, hay una falta de explicabilidad.

Esta falta de explicabilidad hace que no podamos usar de manera directa la salida del modelo. Incluso si la salida es un plan perfecto, no tenemos ninguna certeza de que lo sea. Es por esto que no podemos confiar ciegamente en la salida del modelo y usarla directamente como un plan válido. A lo largo del trabajo, exploramos distintas formas de aprovechar la información que proporciona el modelo, y sabiendo que puede contener errores buscamos formas de minimizar el impacto de los errores en la asignación final de probabilidades a los operadores.

6.2. Análisis de resultados

En este trabajo se presentaron los experimentos más prometedores que realizamos, pero muchos quedaron afuera. Por ejemplo, inicialmente se probaron otros dos modelos Transformer: uno que no separaba los nombres de los números durante la tokenización (para el cual fue necesario aplicar data augmentation y así mitigar el problema de tokens desconocidos) y otro que, en lugar de aprender los embeddings desde cero, utilizaba un modelo preentrenado de embeddings específicos para operadores. El modelo que mejor desempeño mostraba en experimentos iniciales fue el que decidimos utilizar para seguir experimentando, y es el que utilizamos en los resultados presentados.

Además, es importante considerar los tiempos de cómputo involucrados en cada etapa del proceso. Como se detalló en la Sección 5.4, entrenar un modelo durante 50 épocas requería entre 6 y 7 horas, optimizar los hiperparámetros del modelo llevaba aproximadamente 4 días, y ejecutar un experimento con una configuración en particular sobre todo el conjunto de validación tardaba en promedio 40 minutos. Estas limitaciones de tiempo nos llevaron a priorizar ciertos experimentos y configuraciones, buscando siempre el mejor desempeño posible.

Para el dominio Satellite, los resultados óptimos se encontraron con la configuración presentada en el Cuadro 6.1.

Datos de Entrada	**Configuración del Modelo**	**Generación de testigos**	**Distancia Final**
Hechos Relajados en orden de aparición con todos los pares (2054 pares de entrenamiento)	$N_{Encoder}$: 2 $N_{Decoder}$: 6 d: 128 h: 2	Greedy Decode y Beam Search (tamaño 7)	Distancia promedio y porcentaje de objetos no presentes en los hechos relajados

Cuadro 6.1: Tabla de configuración con resultados óptimos

Al usar esta configuración, se alcanzó un PUO promedio de 0,301 sobre los problemas de validación, y más de la mitad de los problemas alcanzan un PUO mayor a 0,3. Esto significa que en estos problemas, se descarta al menos el 30 % de los operadores del problema, y se mantienen los operadores necesarios para resolver el problema. Si bien pareciera no ser mucho, en problemas con una gran cantidad de operadores, descartar el 30 % mediante un proceso que no lleva tanto tiempo, es una mejora segura.

El experimento presentado en la Sección 5.5 mostró que incluso pequeñas cantidades de ruido en el conjunto de testigos pueden degradar significativamente el PUO. Sabiendo que el modelo de traducción no es perfecto y que los testigos se obtienen a partir de

este modelo, en los demás experimentos se probaron distintas estrategias para reducir el impacto de los testigos ruidosos.

En los experimentos presentados en las Secciones 5.6 y 5.8, se demostró que diversificar el conjunto de testigos al incluir más traducciones del modelo mejora el PUO. Esto ocurre porque considerar múltiples traducciones permite no depender de una única traducción potencialmente ruidosa.

De manera similar, en el experimento de la Sección 5.7, se observó que al considerar únicamente la menor distancia con algún testigo, se corre el riesgo de que un operador reciba una probabilidad alta solo por su similitud con un único testigo ruidoso. En cambio, al emplear la distancia promedio con todos los testigos, se amortigua el impacto de los testigos ruidosos al considerar el contexto general proporcionado por todos ellos.

El experimento presentado en la Sección 5.9 mostró que usar el porcentaje de objetos no presentes en los hechos relajados permitió corregir en cierta medida la salida del modelo, descartando operadores que no podrían aparecer según el contexto directo de los hechos relajados antes de ser procesados por el modelo.

Finalmente, en los experimentos presentados en las Secciones 5.10 y 5.11, se ve el impacto que tienen los datos de entrenamiento en el rendimiento. Al utilizar los hechos relajados en orden de aparición, estamos usando datos de entrada para el modelo con más información que los hechos relajados por orden alfabético, obteniendo así mejor PUO. Por otro lado, el uso de un conjunto de entrenamiento más grande, incluso si contiene datos ruidosos, muestra mejores resultados: el modelo es capaz de extraer información útil sobre la estructura y la gramática del problema a pesar del ruido. Además, en el experimento presentado en la Sección 5.12, se observa que ni siquiera optimizando los hiperparámetros para el conjunto de entrenamiento con menos ruido se logra superar el PUO obtenido con el conjunto de entrenamiento más grande.

6.3. Trabajos futuros

Una desventaja de la propuesta presentada es que podría descartar acciones necesarias para resolver un problema, y por lo tanto el planner podría no encontrar un plan cuando sí hay una solución utilizando acciones que no fueron instanciadas. Vemos que grounding y searching son dos etapas muy distintas, cuando podría usarse de alguna forma la información brindada por el planner (por ejemplo, que no se encontró un plan con las acciones brindadas) para mejorar el modelo de traducción. Podría pensarse un enfoque basado en aprendizaje por refuerzo, donde el modelo ajusta su comportamiento en función del desempeño del planner.

Otra posible línea de investigación, es cambiar la función usada para asignar la distancia individual entre un operador y un testigo. En este trabajo, la distancia individual se definió simplemente como la cantidad de tokens de diferencia, pero se podría explorar el uso de embeddings para representar operadores. Si los embeddings logran capturar las similitudes semánticas entre operadores, la distancia entre ellos podría calcularse a partir de la distancia en el espacio de embeddings.

Al elegir la función de distancia para usar, observamos que al usar la distancia mínima, la cantidad de distancias distinas eran pocas: si los testigos de un determinado tipo tienen n tokens, a los operadores de ese tipo se les asigna como distancia un número entero entre 0 y n. Esto llevaba a que la distribución de distancias (y entonces, de las probabilida-

des) fiera muy discreta. Como consecuencia, al instanciar todos los operadores con una probabilidad mayor a un cierto umbral, se terminaban incluyendo grandes cantidades de operadores, ya que todos estaban concentrados en pocos valores posibles. En contraste, la distancia promedio permitió una mayor variabilidad en los valores asignados y una mejor separación entre operadores.

Dado que la función de distancia impacta directamente en la asignación de probabilidades y en la selección final de operadores, un trabajo futuro sería analizar en mayor profundidad sus propiedades matemáticas, y diseñar la función prestando más atención a la distribución de probabilidad que genera, por ejemplo.

También sería interesante evaluar esta estrategia en distintos dominios, no solo Satellite como se hizo en este trabajo. Esto permitiría determinar si las conclusiones a las que se llegaron son específias para este dominio si es una propuesta que puede adaptarse a otros. Además, sería útil probarla en problemas más complejos.

Otro aspecto a mejorar es la construcción de un conjunto de entrenamiento más limpio, es decir, que no asocie hechos relajados con planes incorrectos. Contar con un conjunto de datos más grande y mejor curado podría mejorar el desempeño del modelo al aprender de ejemplos más representativos.

Además, sería conveniente correr experimentos en una máquina con mayor capacidad de VRAM para permitir la generación de más traducciones con Beam Search y evaluar su impacto en el desempeño final.

Notar también que el modelo de traducción no incorpora explícitamente las reglas gramaticales del dominio (por ejemplo, que un operador siempre empieza con el nombre de la acción) ni las restricciones de tipo. Si bien en los experimentos que realizamos el modelo no cometió errores gramaticales evidentes, podría ser interesante integrar estas reglas de manera más explícita. Una posibilidad sería aplicar restricciones durante la generación de la secuencia de salida, de modo que el siguiente token elegido no solo tenga alta probabilidad según el modelo, sino que también sea válido dentro de la gramática.

Por último, sería interesante explorar nuevas formas de aprovechar las probabilidades generadas por el modelo. Por ejemplo, si hay dos operadores posibles para empezar el plan, se podría comparar la probabilidad asignada por el modelo a los tokens iniciales de cada opción. Luego, para determinar el segundo operador, el modelo podría recibir como entrada los hechos relajados y la secuencia parcial generada hasta el momento, y asignar probabilidades a los posibles operadores siguientes. Esto sería una integración del modelo Transformer dentro del planner, guiando la búsqueda con información probabilística.

Bibliografía

Akiba, T., Sano, S., Yanase, T., Ohta, T. and Koyama, M. (2019), *Optuna: A Next-generation Hyperparameter Optimization Framework*, CoRR abs/1907.10902.
URL: http://arxiv.org/abs/1907.10902

Alammar, J. (2018), *The Illustrated Transformer*. Accessed: 2024-12-04.
URL: http://jalammar.github.io/illustrated-transformer/

Areces, F., Ocampo, B., Areces, C., Domínguez, M. and Gnad, D. (2023), *Partial grounding in planning using small language models*, in Proceedings of the 2023 Workshop on Knowledge Engineering for Planning and Scheduling.
URL: https://icaps23.icaps-conference.org/program/workshops/keps/KEPS-23_paper_1243.pdf

Ba, J. L., Kiros, J. R. and Hinton, G. E. (2016), *Layer Normalization*.
URL: https://arxiv.org/abs/1607.06450

Corrêa, A. B., Pommerening, F., Helmert, M. and Francès, G. (2020), *Lifted Successor Generation Using Query Optimization Techniques*, Proceedings of the International Conference on Automated Planning and Scheduling 30(1), 80–89.
URL: https://ojs.aaai.org/index.php/ICAPS/article/view/6648

Erol, K., Nau, D. S. and Subrahmanian, V. (1995), *Complexity, decidability and undecidability results for domain-independent planning*, Artificial Intelligence 76(1), 75–88. Planning and Scheduling.
URL: https://www.sciencedirect.com/science/article/pii/000437029400080K

Fikes, R. E. and Nilsson, N. J. (1971), *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence 2(3–4).
URL: https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/strips.pdf

Fox, M. and Long, D. (2011), *The 3rd International Planning Competition: Results and Analysis*, CoRR abs/1106.5998.
URL: http://arxiv.org/abs/1106.5998

Geffner, H. and Bonet, B. (2013), *A Concise Introduction to Models and Methods for Automated Planning*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Springer, Cham.
URL: https://doi.org/10.1007/978-3-031-01564-9

Ghallab, M., Nau, D. and Traverso, P. (2004), *Automated Planning: Theory and Practice*, Morgan Kaufmann, San Francisco, CA.

Goodfellow, I., Bengio, Y. and Courville, A. (2016), *Deep Learning*, MIT Press, Cambridge, MA.
URL: https://www.deeplearningbook.org/

Hoffmann, J. (2003), *Utilizing Problem Structure in Planning, A Local Search Approach*, Vol. 2854.

Hoffmann, J. and Nebel, B. (2011), *The FF Planning System: Fast Plan Generation Through Heuristic Search*, CoRR abs/1106.0675.
URL: http://arxiv.org/abs/1106.0675

Kingma, D. P. and Ba, J. (2015), *Adam: A Method for Stochastic Optimization*, in Y. Bengio and Y. LeCun, eds, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.
URL: http://arxiv.org/abs/1412.6980

Knight, S., Rabideau, G., Chien, S., Engelhardt, B. and Sherwood, R. (2001), *Casper: space exploration through continuous planning*, IEEE Intelligent Systems 16(5), 70–75.

Lachaux, M., Rozière, B., Chanussot, L. and Lample, G. (2020), *Unsupervised Translation of Programming Languages*, CoRR abs/2006.03511.
URL: https://arxiv.org/abs/2006.03511

Lauer, P., Torralba, A., Fišer, D., Höller, D., Wichlacz, J. and Hoffmann, J. (2021), *Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning*, in Z.-H. Zhou, ed., Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, International Joint Conferences on Artificial Intelligence Organization, pp. 4119–4126. Main Track.
URL: https://doi.org/10.24963/ijcai.2021/567

McCulloch, W. S. and Pitts, W. (1943), *A logical calculus of the ideas immanent in nervous activity*, The Bulletin of Mathematical Biophysics 5(4), 115–133.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. and Wilkins, D. (1998), *PDDL - The Planning Domain Definition Language - Version 1.2*, Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control.
URL: https://www.cs.cmu.edu/~mmv/planning/readings/98aips-PDDL.pdf

Minsky, M. (1968), *Semantic Information Processing*, MIT Press.

Mitchell, T. M. (1997), *Machine Learning*, 1st edn, McGraw-Hill, New York.

Om, H. (2023), *Word Embedding*. Accessed: 2025-02-05.
URL: https://medium.com/@hari4om/word-embedding-d816f643140

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. (2019), *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, CoRR abs/1912.01703.
URL: http://arxiv.org/abs/1912.01703

Richter, S., Westphal, M. and Helmert, M. (2011), *LAMA 2008 and 2011 (planner abstract)*, in IPC 2011 Planner Abstracts, pp. 50–54.
URL: https://ai.dmi.unibas.ch/papers/richter-et-al-ipc2011.pdf

Ridder, B. and Fox, M. (2014), *Heuristic Evaluation Based on Lifted Relaxed Planning Graphs*, Proceedings of the International Conference on Automated Planning and Scheduling 24(1), 244–252.
URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13650

Russell, S. J. and Norvig, P. (2020), *Artificial Intelligence: A Modern Approach*, 4th edn, Pearson, Hoboken, NJ.

Sohrabi, S., Katz, M., Hassanzadeh, O., Udrea, O., Feblowitz, M. D., Riabov, A. and Weng, P. (2019), *IBM Scenario Planning Advisor: Plan recognition as AI planning in practice*, AI Communications 32(1), 1–13.
URL: https://doi.org/10.3233/AIC-180602

Vallati, M., Magazzeni, D., De Schutter, B., Chrpa, L. and McCluskey, T. (2016), *Efficient Macroscopic Urban Traffic Models for Reducing Congestion: A PDDL+ Planning Approach*, Proceedings of the AAAI Conference on Artificial Intelligence 30(1).
URL: https://ojs.aaai.org/index.php/AAAI/article/view/10399

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I. (2017), *Attention Is All You Need*, CoRR abs/1706.03762.
URL: http://arxiv.org/abs/1706.03762

Williams, R. J. and Zipser, D. (1989), *A learning algorithm for continually running fully recurrent neural networks*, Neural Computation 1(2).