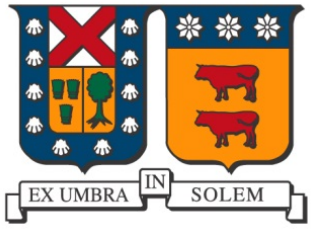
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAISO - CHILE



“SISTEMA RECOMENDADOR ENFOCADO A BARES MEDIANTE TÉCNICAS DE MACHINE LEARNING PARA PROYECTO CROPY”

FELIPE CISTERNAS ALVAREZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Valle Vidal
Profesor Correferente: Ricardo Ñanculef

Octubre - 2023

DEDICATORIA

Dedicado a mi familia, que me ha permitido poder estudiar, cumplir mis sueños y crecer como persona, también dedicarlo a mis amigos que me han brindado apoyo y motivación y por ultimo pero no menos importante a mis profesores que me han acompañado en estos años de estudio, gracias a ellos por compartir su conocimiento y motivarme a seguir aprendiendo.

AGRADECIMIENTOS

Agradecer a mis profesores, colegas y amigos que me han brindado su conocimiento y ayuda para poder desarrollar esta memoria

- Dr. Carlos Valle Vidal, Universidad Técnica Federico Santa Maria
- Dr. Ricardo Ñanculef, Universidad Técnica Federico Santa Maria
- Dr. Hector Allende, Universidad Técnica Federico Santa Maria
- Dra. Raquel Pezoa, Universidad Técnica Federico Santa Maria
- Dr. Denis Parra, Pontificia Universidad Catolica
- Diego Cattarinich
- Bruno Prieto
- Lucas Galindo
- Diego Quezada
- Antonio Martinez
- Jose Musso

RESUMEN

Resumen— Los sistemas de recomendación son medios valiosos para que los usuarios hagan frente a la sobrecarga de información y les ayuden a tomar mejores decisiones. En esta memoria se propone un modelo basado en redes neuronales y Transformers para recomendar distintos bebestibles y platos a usuarios que frecuentan bares. El sistema recomendador se desarrolló en TensorFlow y se lanzó a producción mediante el uso de contenedores Docker. La solución fue validada frente a distintos experimentos para escoger la mejor arquitectura e hiperparámetros, utilizando datos reales de centros de ventas de Viña del Mar, Chile. Un buen modelo recomendador puede recomendar y ayudar a sus usuarios a tener una mejor experiencia en el sector gastronómico, así como también lo pueden utilizar los administradores para incentivar el consumo de productos dentro de los bares, proporcionando valor tanto para clientes como administradores de Bares y Restaurantes.

Palabras Clave— Sistemas Recomendadores; Aprendizaje Automático; Gastronomia; Redes Neuronales; Ciencia de datos.

ABSTRACT

***Abstract*—** Recommender systems are valuable means for users to cope with information overload and help them make better decisions. In this paper, a model based on neural networks and Transformers is proposed to recommend different drinks and dishes to users who frequent bars. The recommender system was developed on TensorFlow and released to production using Docker containers. The solution was validated against different experiments to choose the best architecture and hyperparameters, using real data from sales centers in Viña del Mar, Chile. A good recommender model can recommend and help its users have a better experience in the gastronomic sector, as well as can be used by administrators to encourage the consumption of products within bars, providing value for both customers and Bar or Restaurants administrators

***Keywords*—** Recommenders Systems; Machine Learning; Gastronomy; Deep Learning; Data Science.

GLOSARIO

API: Application Programming Interface.
BERT: Bidirectional Encoder Representations from Transformers.
CPU: Central Processing Unit.
CSV: Comma Separated Values.
CV: Computer Vision.
DCN: Deep & Cross Network.
DI: Departamento de Informática.
DL: Deep Learning.
EDA: Exploratory Data Analysis.
GPT: Generative Pretrained Transformer.
GPU: Graphic Processing Unit.
IA: inteligencia Artificial.
JSON: JavaScript Object Notation.
KNN: K-nearest neighbors.
LLM: Large Language Model.
LN: Layer Normalization.
LR: Learning Rate.
MAE: Mean Absolute Error.
ML: Machine Learning.
MSE: Mean Squared Error.
NLP: Natural Language Processing.
NMT: Neural Machine Translation.
PCA: Principal Component Analysis.
PDF: Portable Document Format.
PUC: Pontificia Universidad Catolica.
QR: Quick Response.
RAM: Random-Access Memory.
RMSE: Root Mean Squared Error.
SGD: Stochastic gradient descent.
OS: Operative System.
SVD: Singular value decomposition.
SVM: Support Vector Machines.
TFRS: TensorFlow Recommender Systems.
TFX: TensorFlow Extended.
TPU: Tensor Processing Unit.
T-SNE: T-Distributed Stochastic Neighbor Embedding.
UTFSM: Universidad Técnica Federico Santa María.
XAI: Explainable Artificial Intelligence.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La pandemia del COVID-19, ha tenido un impacto significativo en la industria gastronómica, especialmente en los bares y restaurantes. La reducción de aforos, los largos tiempos de espera y otros factores han dificultado la satisfacción de los clientes y, por lo tanto, han afectado las ventas de los negocios. En este contexto, surge la necesidad de encontrar soluciones innovadoras que permitan mejorar la experiencia de los comensales y, al mismo tiempo, aumentar la rentabilidad de los establecimientos.

En esta memoria, para optar al título de Ingeniero Civil en Informática, se aborda el problema de la baja en las ventas de los bares y se propone un sistema recomendador para ofrecer una experiencia personalizada a los comensales y administradores de bares.

Los sistemas de recomendación son una tarea bastante compleja al tener que predecir el comportamiento y gustos de los seres humanos. Existen muchos factores tanto externos como internos que pueden afectar al momento de querer consumir un alimento, desde la hora del día, el estado de ánimo, pedidos pasados, etc. Es por esto que es necesario definir minuciosamente los atributos de entrada, mantener actualizados los algoritmos de recomendación y por último, pero no menos importante, el poder desplegar la solución de manera eficiente para el uso de los clientes.

En esta investigación, se desarrolla un sistema recomendador que utiliza redes neuronales y Transformers para el software Cropy. El modelo recomendador propuesto se enfoca en ofrecer recomendaciones de platos y bebidas a los clientes de los bares, teniendo en cuenta sus preferencias y gustos personales. Además, el modelo también puede ser utilizado por los administradores de los bares para optimizar la oferta de productos y mejorar la rentabilidad del negocio.

La presente memoria pertenece al área de Inteligencia Computacional del Departamento de Informática de la Universidad Técnica Federico Santa María y consta de 5 capítulos.
En el capítulo 1 se define exhaustivamente el problema, se describe el contexto actual de la industria gastronómica y se identifican las principales dificultades que enfrentan los bares.
En el capítulo 2 se discute el marco conceptual, proporcionando al lector el contexto necesario sobre sistemas recomendadores, inteligencia artificial, machine learning, redes neuronales, así como las arquitecturas y técnicas utilizadas junto con sus fundamentos matemáticos.
En el capítulo 3 se propone la solución, se analizan los datos, se describe la arquitectura y se detallan los pasos seguidos junto con las herramientas utilizadas para su implementación.
En el capítulo 4 se valida la solución, se presentan la metodología y los experimentos realizados junto con los resultados obtenidos, se discute la complejidad del sistema recomendador, su explicabilidad y despliegue a producción para su funcionamiento en el mundo real.
Finalmente, en el capítulo 5 se plantean las conclusiones de la presente memoria, considerando los alcances, limitaciones de la solución planteada y posibles mejoras del modelo.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

En esta sección se define el problema abordado en la presente investigación. Se describe el contexto, el funcionamiento actual del proceso que se busca mejorar, las principales dificultades actuales y finalmente los objetivos y alcance de la memoria.

1.1. Contexto

Durante la pandemia, muchos negocios sufrieron una gran baja en sus ventas, lo que les dificulto mantenerse a flote donde muchos negocios cerraron, una de las industrias más afectadas por la pandemia fue la industria gastronómica [Saravia, 2021], ya que con aforos reducidos, largos tiempos de espera, y muchos otros factores, simplemente no podían satisfacer de buena manera a todos los clientes que acudían a los locales.

Los Bares fueron un segmento muy damnificado, ya que durante la pandemia y el estallido social en chile hubo mucho tiempo de toque de queda, donde la gente no podía salir a compartir en los bares, esto generó una gran disminución en sus clientes, sumando que los bares ya venían con grandes pérdidas[M, 2021]. Una vez se comenzaron a disminuir las restricciones de la pandemia, las personas llenaban los bares haciendo que se formaran filas enormes donde se tenía que esperar por horas para poder ingresar al local, esto generó un gran descontento en los comensales, una vez adentro los meseros y el equipo de cocina no estaba preparado para poder afrontar tantos clientes y era bastante común que los tragos y comidas se fueran acabando rápidamente, creando una experiencia nefasta para los comensales.

La gastronomía está en un proceso de digitalización, donde ahora para poder revisar la carta, la tenemos que escanear mediante un código QR a través de nuestro celular, pero esta digitalización aún está naciendo.

1.1.1. Cropy

Cropy es un servicio enfocado a bares, que busca mejorar la experiencia de usuario de los comensales que frecuentan el bar, como también mejorar la administración de este mismo negocio. Para un administrador de bar, le es sumamente importante poder gestionar su personal y su inventario de manera óptima para poder atender a la mayor cantidad de personas y poder así obtener la mayor cantidad de beneficios para su local, sabemos que la cantidad de variables en juego son muchísimas y no se puede tener control o análisis sobre todas ellas al mismo tiempo.

Nos encontramos en una era tecnológica donde la base de todos los servicios y software que existen a día de hoy son los datos, el poder recolectar, procesar y generar información mediante estos datos es lo que hace sobresalir a las grandes empresas tecnológicas, solo basta mirar el mercado y ver como las empresas tecnológicas que trabajan sus datos son las más grandes y las que dominan la industria, empresas como Meta, Google, Microsoft, Apple, etc. Si bien los datos son la base de todos los sistemas, si no somos capaces de poder analizar estos datos y extraer información importante de los mismos, no vamos a lograr nada, es por esto que se han realizado importantes avances en el área de la inteligencia artificial y sobre todo en el área del machine learning, una línea informática donde se busca que los modelos o algoritmos puedan aprender y mejorarse a sí mismos. La masividad de los datos y los avances en el campo de la inteligencia artificial nos han permitido crear soluciones nunca antes vistas, como lo puede ser la conducción autónoma, detección de fraudes bancarios, análisis de patrones en retailers, etc. Estas soluciones permiten poder obtener una eficiencia y optimización gigantescas del negocio, por lo que es esencial utilizar y trabajar los datos que tenemos a nuestra disposición.

1.2. Problemas

1.2.1. Problema 1: Segmentación de usuarios

Este problema no es único de los bares, sino de la industria gastronómica en general, de hecho se puede extender a muchas otras industrias, el poder segmentar a tus clientes y saber sus preferencias es una ventaja enorme por sobre la competencia, pudiendo ofrecer una mejor experiencia en tu negocio, personalizada para cada usuario con tal de poder atraer a la mayor cantidad de clientes y beneficios. Un claro ejemplo es Cornershop, las grandes multitiendas y supermercados, donde estos organizan sus productos de tal manera que la gente gaste más dinero en su negocio, analizando los productos que más se venden en conjuntos, creando ofertas, etc.

1.2.2. Problema 2: Recopilación de Métricas

Cropy se enfoca en mejorar la experiencia de usuario de comensales que asisten a bares, pero si bien la experiencia de usuario mejora con nuestros sistemas en tiempo real, se queda lejos de ser una experiencia totalmente personalizada para cada usuario. Los clientes que asisten a un bar con Cropy pueden elegir y pedir todo desde sus celulares, pero es una experiencia bastante estándar, todos ven las mismas cosas aun cuando cada persona tiene gustos e intereses distintos, por ejemplo a una persona le pueden gustar los tragos más dulces mientras que a otro le pueden gustar los tragos más ácidos, esta información sobre el gusto de los clientes es un dato muy útil sobre todo para los administradores de los bares, ya que con esta información ellos pueden saber cuál es el trago más vendido y saber si les conviene incluir

otras nuevas preparaciones basándose en el gusto de sus comensales.

1.2.3. Problema 3: Optimización de Procesos

Uno de los puntos más importantes es poder optimizar los procesos del Bar, como por ejemplo la venta de tragos basándose en ciertos parámetros como pueden ser la hora, la temperatura, la fecha, los pedidos anteriores, etc. De esta forma, el Bar puede saber en todo momento el gusto de sus comensales y así preparar su negocio, su personal, su inventario para poder afrontar de la mejor manera la demanda maximizando sus ganancias.

1.2.4. Problema 4: Experiencia Personalizada

Otro problema que resaltan muchos los meseros que hemos entrevistado como equipo detrás de Cropy, es que la gente no sabe qué pedir y a esto le suman que en ocasiones ya no hay stock de ciertos productos, genera que el comensal pierda mucho tiempo en decidir que va a consumir y esto le hace perder tiempo al mesero y retrasa todo el flujo de clientes para un Bar. Cropy soluciona la gestión del bar, pero necesita un buen sistema recomendador para poder apoyar las decisiones del negocio, agrupando y analizando a sus clientes, de tal manera que el bar pueda ofrecer promociones, pueda estar tener su inventario siempre preparado para satisfacer sus clientes, puedan apuntar a públicos objetivos y crear una experiencia más agradable y personalizada para sus clientes, donde a cada cliente se le muestre una gama de productos acorde a sus pedidos pasados y que con el tiempo el algoritmo pueda aprender efectivamente los gustos del consumidor, detectando patrones y relaciones que los humanos pasamos por alto.

1.2.5. Problema 5: Explicabilidad

Por último, otro problema fundamental, es la poca explicabilidad de los modelos de machine learning, es verdad que detrás de estos modelos y algoritmos hay una base matemática y estadística muy fuerte, pero el usuario final de estos modelos, un gerente de bar, un usuario que no sabe qué pedir, no va a tener esa base teórica, por lo que hacer el modelo lo más explicable posible, mostrar métricas acordes, simples y concisas son una tarea fundamental, muchas investigaciones de inteligencia artificial remarcan enormemente lo importante que es construir un software accesible para todos los usuarios, que no se necesite de un experto para poder utilizarlo y así poder abarcar un público objetivo aún mayor.

1.3. Actores

Para este problema el número de actores que interactúan son bastante pocos, pero eso no significa que haya que menospreciar el problema, de hecho al tener una cantidad reducida de actores nos permite centrarnos mejor y más detalladamente en cada uno de ellos, para suplir sus necesidades, en la siguiente tabla (Ver Tabla 1) se puede observar la descripción, el dolor y el valor de cada actor.

Actores	Descripción	Dolor	Valor
Comensales	Clientes que asisten a los Bares	Largos tiempos de espera para poder consumir sus productos	Ingresos para el Bar, a mayor cantidad de clientes mayores ingresos
Meseros	Personal que atiende el Bar	Trabajo Saturado y perdida de tiempo cuando clientes no se deciden que pedir	Suplir necesidades de Comensales, Son los encargados de entregar una buena experiencia dentro del Bar
Administrador	Encargado de gestionar el inventario y el personal del Bar	Difícil trazabilidad de variables para gestionar de mejor manera el Bar	Optimización de recursos, es el encargado de gestionar los procesos para que el Bar maximice sus ventas

Tabla 1: Tabla de Actores del Problema

1.4. Causas y Consecuencias

Existen una serie de causas que originan todos estos problemas y estas causas a su vez generan consecuencias, las cuales mediante esta memoria queremos solucionar. Se resumirán las causas y consecuencias más importantes a continuación, pero para ver una mirada más detallada de las mismas se recomienda analizar el **Árbol del Problema** (Ver figura 1).

1.4.1. Causas más importantes

- Atención al consumidor Deficiente.
- Largos tiempos de espera.
- Gestión deficiente del personal.
- Se agota el stock de productos.

1.4.2. Consecuencias más importantes

- Clientes no vuelven otra vez al Bar.
- Empleados Descontentos.
- Mala reputación del local.
- Bares no venden el 100 % de sus productos.

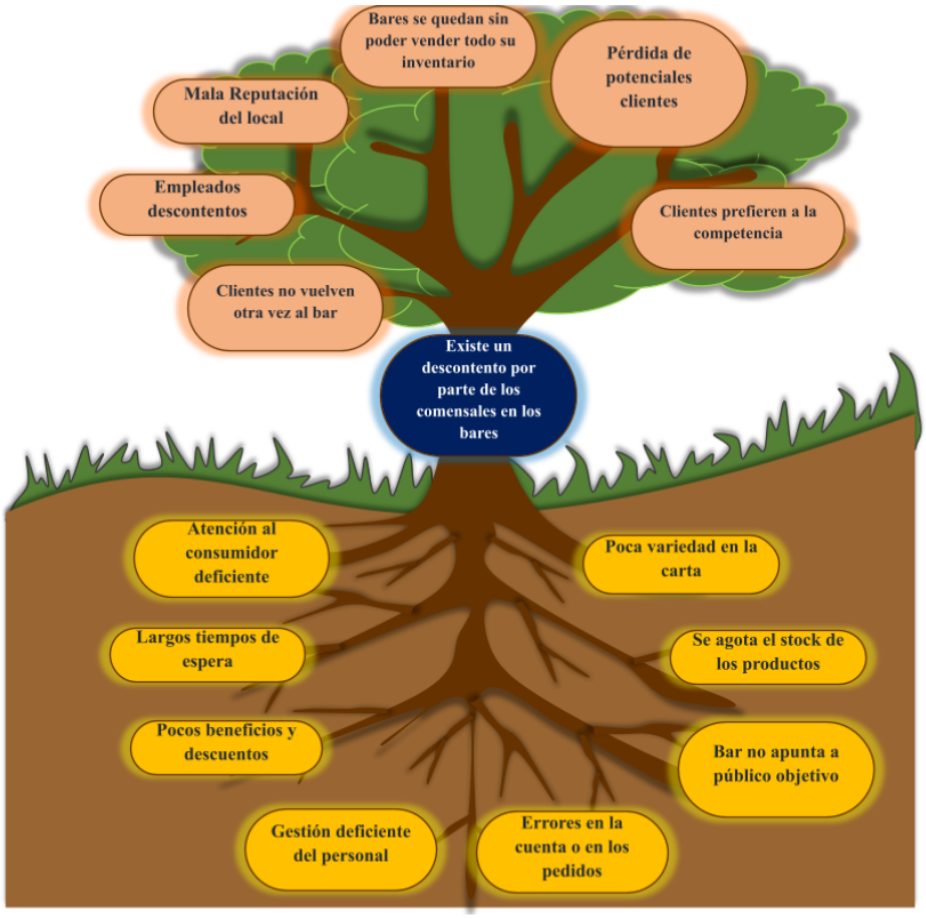


Figura 1: Árbol Del Problema.
(Las raíces son las causas de los problemas, el tronco es el problema a grandes rasgos y las hojas son las consecuencias de los problemas.)
Fuente: Elaboración Propia.

1.5. Estado Actual

Actualmente, la mayoría de los servicios gastronómicos, no solo Bares, utilizan la tecnología de códigos QR para mostrar un menú estático en formato PDF, donde la gente puede ver la carta mientras espera a ser atendida. Cropy ofrece interactividad en tiempo real para los comensales de un Bar, permitiéndoles ver y pedir tragos o comestibles desde su celular, viendo en todo momento que hay y que no hay mediante un menú que se actualiza en tiempo real, también les permite poder llamar a los meseros ante alguna necesidad, separar las cuenta entre los distintos usuarios y pagar, todo desde el celular, pero aun con todas estas funcionalidades no podemos ofrecer una experiencia totalmente personalízale para cada cliente.

1.6. Competencias

A día de hoy no existe una competencia directa a Cropy y el modelo recomendador que se va a desarrollar, pero si existen recomendadores en este caso de cervezas y tragos muy parecidos, la diferencia recae en que estas aplicaciones apuntan más a ser una especie de red social o foro donde existe una base de datos de cervezas y tragos y la gente los puede calificar y dar su opinión, algunas de estas aplicaciones indican donde puedes ir a beber los tragos que te gustan, pero no es un software propio que implemente el bar, por lo que no se puede considerar una competencia directa. Algunas de estas aplicaciones son:

- BeerAdvocate
- Beer Style Guidelines
- Ratebeer

1.7. Objetivos y Alcances

El objetivo principal de esta memoria es poder desarrollar un modelo recomendador lo bastante robusto que mezcle las distintas características de los datos para ofrecer una experiencia lo más personalízale posible, todo esto bajo una serie de objetivos como

- Tiempos aceptables de inferencia
- Explicabilidad del modelo
- Pipeline automatizada de datos para entrenamiento e inferencia
- Escalabilidad del modelo

Todo esto con el fin de poder desarrollar una herramienta útil tanto como para los comensales de un Bar, como para los administradores de los mismos. También se busca la mejor generalización de la problemática planteada, esto quiere decir que es necesario que el modelo se pueda adaptar a los distintos bares y sea capaz de producir resultados precisos en cada uno de ellos.

CAPÍTULO 2
MARCO CONCEPTUAL

En esta sección se plantea el marco conceptual de la presente investigación, definiendo exhaustivamente los conceptos fundamentales en los cuales se sostiene la solución planteada en el capítulo 3.

2.1. Sistemas Recomendadores

Los sistemas de recomendación son herramientas y técnicas de software que proporcionan sugerencias de elementos que pueden ser de utilidad para un usuario. Las sugerencias proporcionadas por un sistema de recomendación tienen como objetivo ayudar a sus usuarios en varios procesos de toma de decisiones, como qué artículos comprar, qué música escuchar o qué noticias leer. Los sistemas de recomendación son medios valiosos para que los usuarios hagan frente a la sobrecarga de información y les ayuden a tomar mejores decisiones [Ricci *et al.*, 2011].

Antes de entrar más en detalle en los sistemas recomendadores es necesario definir algunos términos:

- Ítem: también conocidos como documentos, son las entidades que un sistema recomienda a un usuario, en el caso del Bar son los tragos o comidas, en el caso de YouTube serían los videos, en el caso de Spotify serían las canciones, etc.
- Query: también conocido como contexto, es la información que el sistema utiliza para realizar las recomendaciones, las querys o consultas pueden ser combinaciones de la información del usuario, su ID, los ítems con los que interactuó previamente, o también puede ser información adicional externa, como la fecha, hora, clima, etc.

En su forma más simple, los sistemas de recomendación cumplen dos tareas, primero retornar una lista de ítems relevantes para el usuario y segundo ordenar dicha lista de ítems según las preferencias del usuario. Para poder realizar estas tareas, los sistemas de recomendación utilizan la información del usuario y también sus interacciones y feedback con los ítems.

Existen distintas formas de resolver un problema de recomendación basándose en el tipo de información que utilizamos para predecir (Ver figura 3) y se dividen en tres grandes grupos: Filtrado en base a contenidos, filtrado en base a colaboraciones y por último enfoques mixtos, también se dividen los sistemas recomendadores en base al feedback (Ver figura 4), rating o calificación que proporciona el usuario y existen dos grandes grupos: feedback implícito y feedback explícito [Aggarwal, 2016].

2.1.1. Métodos de filtrado

Filtros basados en contenidos (Content-Based Filtering)
En este método, el modelo analiza las características de los ítems con los que el usuario ha interactuado en el pasado, y el algoritmo intenta recomendar ítems similares a los que el usuario ha calificado de buena manera. Este método trata las recomendaciones como una clasificación específica y única para cada usuario.

Filtros colaborativos (Collaborative Filtering)
Este método se basa en asumir que usuarios que les gustaron las mismas cosas en el pasado tendrán gustos parecidos en el futuro y les gustarán ítems similares. En este método primero se agrupan a los usuarios con gustos similares y después se recomiendan ítems basándose en las preferencias del grupo. A diferencia de los filtros basados en contenidos acá, las recomendaciones ya no son específicas y únicas para cada usuario, sino que se tratan dependiendo del clúster de usuarios. Primero se agrupa a los usuarios según sus gustos con algún algoritmo de clustering o modelos de **Unsupervised Learning**, después se genera una matriz entre los usuarios y los ítems, esta matriz va a ser muy dispersa, es decir, habrá muchos datos que no estarán disponibles, ya que no todos los usuarios han interactuado con todos los ítems, es por eso que se busca poder predecir de la manera más precisa esa interacción que tendrá determinado usuario con un ítem basándonos en los gustos de usuarios con gustos similares a él en el pasado, luego se realiza una predicción del rating que el usuario dará a los ítems, esto se suele realizar mediante dos métodos:

1. Suma Ponderada (Weighted Sum):

$$\hat{P_{u,i}} = \frac{\sum_{\text{all similar items, N}}(S(i,N) \cdot R_{u,N})}{\sum_{\text{all similar items, N}} S(i,N)}.$$

   Donde u es un usuario, i es un ítem, N va iterando sobre todos los ítems similares, $S()$ es una función que calcula la similitud entre ítems y $R()$ es la calificación o rating entre un usuario y un ítem.

2. Regresión:

$$\overrightarrow{R'_{u,i}} = \alpha \cdot \overrightarrow{R'_i} + \beta + \epsilon.$$

   Donde α y β son parámetros entrenables que se van aprendiendo con los ratings de los usuarios que han proporcionado sus calificaciones, $\overrightarrow{R'_i}$ es el vector con la información del ítem i y ϵ es el error intrínseco de la regresión.

Por último, se recomienda el ítem con mejor calificación al usuario, basándose en las preferencias de sus vecinos más cercanos.

Dentro del filtrado colaborativo existen varios métodos para resolver el problema, siendo uno de los más famosos el método de **Matrix Factorization** o factorización de matrices, estos algoritmos se basan es descomponer la matriz de interacciones usuario-ítem en dos matrices rectangulares de menor dimensionalidad (Ver figura 2). Estos algoritmos ganaron mucha popularidad con el **Netflix Prize** [Bennett y Lanning, 2007] una competición que organizo Netflix durante los años 2007-2009, ofreciendo 1 millón de dólares al equipo que superara su sistema de recomendaciones para películas, este fue un problema de filtrado colaborativo donde se popularizó el uso de factorización de matrices para resolver el problema.

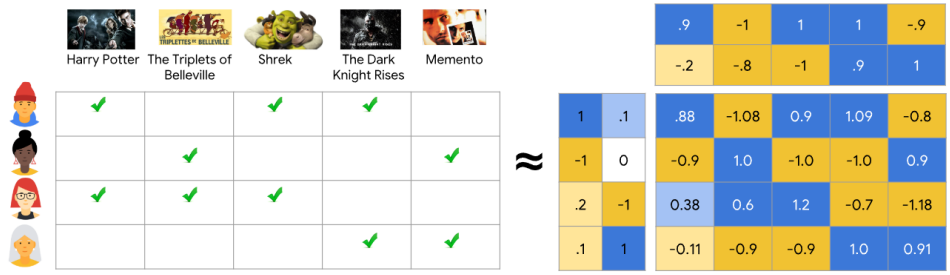


Figura 2: Factorización de matriz Usuario-Pelicula.
Fuente: Google Developers.

Existen una serie de problemas al utilizar filtros colaborativos en los problemas de recomendación, como lo pueden ser:

- **Cold Start (Inicio Fresco):** Para un nuevo usuario o ítem no hay suficiente data ni interacciones que permitan hacer buenas recomendaciones [Datta *et al.*, 2023].
- **Scalability (Escalabilidad):** Hay millones de usuarios y producto los que estos sistemas hacen recomendaciones. Por lo tanto, se necesita una gran cantidad de potencia de cálculo para calcular las recomendaciones.
- **Sparsity (Dispersión):** Los usuarios más activos solo habrán calificado un pequeño subconjunto de la base de datos general. Por lo tanto, incluso los artículos más populares tienen muy pocas calificaciones.

Recomendaciones híbridas

La mayoría de los recomendadores ahora utilizan un método mixto que combina el enfoque de Content-Based Filtering y el enfoque de Collaborative Filtering, ya que cada método tiene sus ventajas y desventajas [Kumar *et al.*, 2021]. Spotify es un buen ejemplo de sistemas híbridos, ya que recomienda música que comparten características con canciones que el usuario ha calificado de mejor manera en el pasado (Content-based Filtering) y también recomienda música basándose en los hábitos de usuarios similares (Collaborative Filtering).

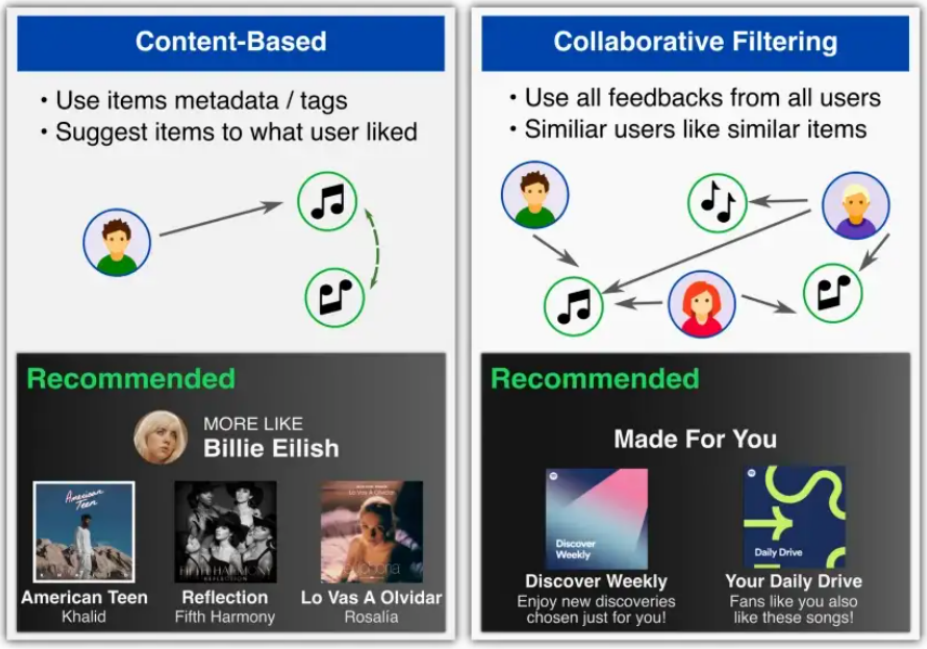


Figura 3: Content-based y Collaborative Filterings.
Fuente: Francesco Casalegno, Towards Data Science.

2.1.2. Tipos de Feedbacks

Feedback Explícito
Este tipo de feedback como su nombre lo indica es un feedback explícito, es decir, el usuario califica explícitamente cuanto le gusta determinado ítem, esta calificación puede ser numérica como por ejemplo estrellas desde el 1 al 5 o cualitativa como por ejemplo utilizar me gusta, me encanta, etc. Es de las formas más fáciles de tratar problemas de recomendación, ya que el usuario nos entrega explícitamente cuanto le gusta ese ítem.

Feedback Implícito
Muchas veces no vamos a tener la valoración del usuario, ya sea porque el usuario no sé dé el tiempo de calificar los ítems o porque el sistema no tiene implementadas las calificaciones, es por eso que no podemos saber si el usuario que interactuó con un ítem, tuvo una buena o mala experiencia, es por esto que se analizan los patrones de los usuarios frente a los ítems, por ejemplo analizando los tiempos que cada usuario visualiza cierto ítem o la cantidad de clics frente al ítem, teniendo un registro histórico de las interacciones ítem-usuario ya sea en compras para casos de retails, o visualizaciones en caso de películas y música.

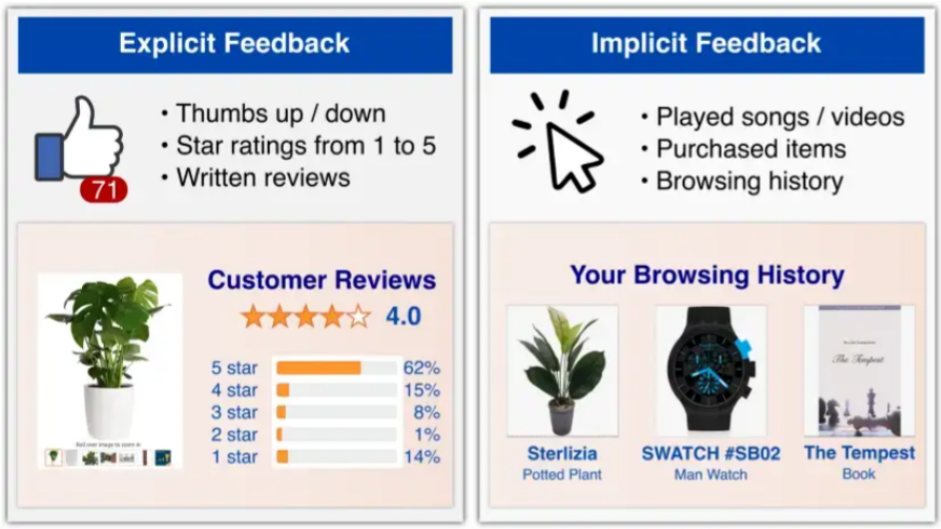


Figura 4: Explicit e Implicit Feedbacks.
Fuente: Francesco Casalegno, Towards Data Science.

2.1.3. Modelo Matemático

Consideremos una matriz $M_{n,m}$ donde n es la cantidad de usuarios y m la cantidad de ítems. Cada celda M_{ij} representa la calificación o rating que un usuario n le dio a un ítem j. En esta matriz solo han sido calificados ciertos ítems por cada usuario, por lo tanto, vamos a tener una matriz *dispersa*, ya que no conocemos muchos de los valores dentro de ella y la tarea del sistema recomendador es poder predecir estas interacciones faltantes, para esto vamos a factorizar la matriz M tal que:

$$M \approx X \cdot Y^T.$$

donde X es la "Matriz de Usuarios", cuyas filas representan los n usuarios y las columnas representan distintos atributos del usuario, por lo que cada fila de la matriz será un vector del usuario y su información. Y es la "Matriz de ítems", cuyas filas representan los m ítems y las columnas representan distintos atributos de los ítems, al igual que con la matriz de usuarios, cada fila de esta matriz será un vector de los ítems y su información correspondiente. Estos vectores no tienen una dimensión fija y dependen de los atributos que se recolecten, en el caso del usuario puede ser la edad del cliente, el sexo, etc. Y lo mismo para con el vector de cada ítem, a estos vectores los llamaremos *Embeddings*, un vector multidimensional que contiene la representación de la información del usuario o ítem.

$$\begin{aligned} user_i &\equiv X_i & \forall i \in 1, ..., n, \\ item_j &\equiv Y_j & \forall j \in 1, ..., m. \end{aligned}$$

buscamos matrices X e Y cuyo producto punto se aproxime mejor a las interacciones existentes. Vamos a denotar E los pares (i, j) que estén definidos, tal que:

$$M_{ij} \neq \varnothing.$$

Queremos encontrar X e Y tal que minimice el error de reconstrucción de calificación (rating reconstruction error), que en este caso será el **Error cuadrático medio** (MSE)

$$(X, Y) = \underset{X,Y}{\text{argmin}} \sum_{(i,j)\in E} [(X_i)(Y_j)^T - M_{ij}]^2.$$

Agregando un factor de regularización y dividiendo por 2 obtenemos:

$$(X, Y) = \underset{X,Y}{\text{argmin}} \frac{1}{2} \sum_{(i,j)\in E} [(X_i)(Y_j)^T - M_{ij}]^2 + \frac{\lambda}{2}(\sum_{i,k}(X_{ik})^2 + \sum_{j,k}(Y_{jk})^2).$$

Ahora las matrices X e Y pueden ser obtenidas siguiendo un proceso de optimización como lo puede ser el método de **gradiente descendente**. Una vez que la matriz ha sido factorizada, para hacer una nueva recomendación simplemente podemos multiplicar un vector de usuario por cualquier vector de ítem para estimar el rating correspondiente.

Finalmente, podemos notar que el concepto de esta factorización básica se puede extender a modelos más complejos como, por ejemplo, redes neuronales. Esta extensión la hacemos encapsulando el producto punto mediante alguna función de activación como por ejemplo una función logística, así obtenemos un modelo que toma los valores en $[0, 1]$ y se ajusta mejor al problema, en tal caso el modelo a optimizar es el siguiente:

$$(X, Y) = \underset{X,Y}{\text{argmin}} \frac{1}{2} \sum_{(i,j)\in E} [f((X_i)(Y_j)^T) - M_{ij}]^2 + \frac{\lambda}{2}(\sum_{i,k}(X_{ik})^2 + \sum_{j,k}(Y_{jk})^2).$$

donde $f()$ sería la función logística u otra función de activación. Modelos de redes neuronales profundas se utilizan para lograr el estado del arte en sistemas de recomendación complejos (Ver figura 5).

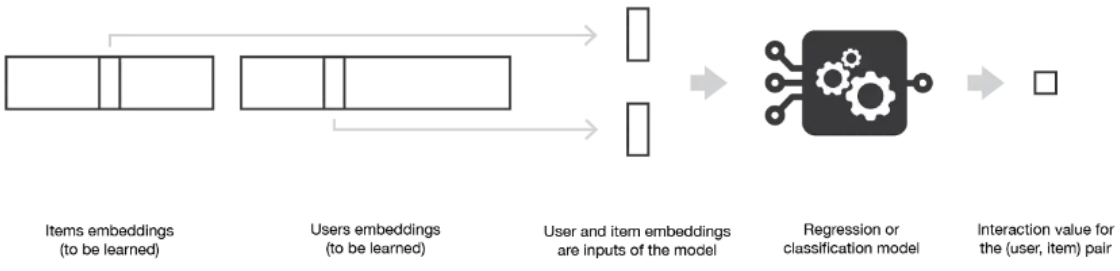


Figura 5: Generalización de factorización de matrices mediante modelos de machine learning.

Fuente: Baptiste Rocca, Towards Data Science.

También podemos utilizar a la propia red neuronal para que aprenda a generar estos embeddings o representaciones tanto de los usuarios como de los ítems, y después calcular la similitud de estos vectores, así cuando un usuario califique de buena manera cierto ítem, ambos vectores estarán más cercanos en el espacio de embeddings, y la tarea de la red neuronal es poder modelar este espacio multidimensional y calcular la distancia o similitud entre vectores.

2.1.4. Metricas

Existen variadas métricas para abordar problemas de recomendación dependiendo lo que estemos evaluando, ya sea el rating, la lista de recomendaciones, etc.

las métricas más tradiciones para evaluar el **Rating** son:

- MAE: Mean Absolute Error

$$\text{MAE} = \frac{\sum_{i=1}^{n} |\hat{r_{ui}} - r_{ui}|}{n}.$$

- MSE: Mean Squared Error

$$\text{MSE} = \frac{\sum_{i=1}^{n} (\hat{r_{ui}} - r_{ui})^2}{n}.$$

- RMSE: Root Mean Squared Error

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{r_{ui}} - r_{ui})^2}{n}}.$$

Por otro lado, si queremos evaluar una lista de recomendaciones podemos utilizar **Accuracy, Precisión y Recall**, si consideramos los elementos recomendados como un conjunto S y los elementos relevantes como el conjunto R tenemos:

$$\text{Accuracy} = \frac{|\text{Recomendados} \cap \text{Relevantes}|}{|\text{Recomendados} \cup \text{Relevantes}|},$$

$$\text{Precisión} = \frac{|\text{Recomendados} \cap \text{Relevantes}|}{|\text{Recomendados}|},$$

$$\text{Recall} = \frac{|\text{Recomendados} \cap \text{Relevantes}|}{|\text{Relevantes}|}.$$

Al aumentar el Recall (la proporción de elementos relevantes) disminuimos la precisión, por lo cual hay un compromiso entre ambas métricas, es por esto que generalmente se utiliza la métrica **F1 Score**

$$F1 = \frac{2 * \text{precisión} * \text{Recall}}{\text{precisión} + \text{Recall}}.$$

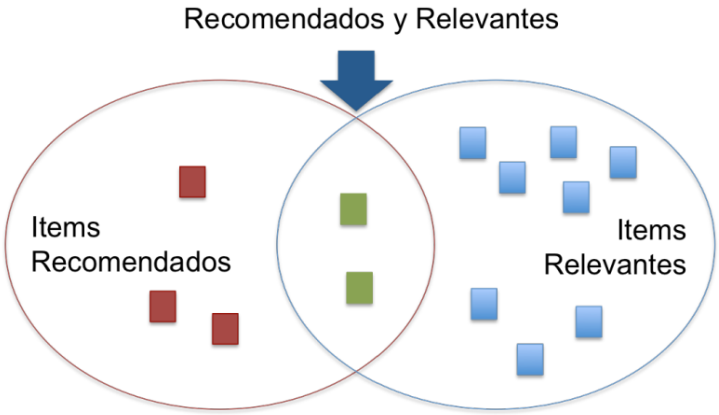


Figura 6: Conjuntos recomendados y relevantes.
Fuente: Denis Parra, PUC.

Más allá de estas métricas, nos puede interesar evaluar otros aspectos como

- **Diversidad:** Los usuarios tienden a estar más satisfechos con las recomendaciones cuando hay una mayor diversidad dentro de la lista
- **Persistencia:** En algunas situaciones, es más efectivo volver a mostrar recomendaciones o permitir que los usuarios vuelvan a calificar elementos que mostrar elementos nuevos.
- **Privacidad:** Los sistemas de recomendación generalmente tienen que lidiar con problemas de privacidad porque los usuarios tienen que revelar información confidencial, la creación de perfiles de usuario mediante el filtrado colaborativo puede ser problemática desde el punto de vista de la privacidad
- **Demografía:** Joeran Beel descubrió que los datos demográficos de los usuarios pueden influir en la satisfacción de los usuarios con las recomendaciones, En su artículo [Beel *et al.*, 2013] muestran que los usuarios mayores tienden a estar más interesados en las recomendaciones que los usuarios más jóvenes.
- **Robustez:** Cuando los usuarios pueden participar en el sistema de recomendación, se debe abordar el problema del fraude.
- **Casualidad:** es una medida de cuán sorprendentes son las recomendaciones, por ejemplo si un usuario va a una panadería, recomendar un pan sería una recomendación acertada, pero no es una buena recomendación, ya que es algo obvio que el usuario comprara pan, en cambio, si le recomendamos algún pastel o algo más distinto, pero aun dentro del ámbito de sus gustos esa si será una buena recomendación.

- **Confianza:** Un sistema de recomendación tiene poco valor para un usuario si el usuario no confía en el sistema. La confianza puede generarse mediante la **explicabilidad** de los modelos de machine learning, un punto muy importante a día de hoy, ya que la mayoría de los modelos son tratados como cajas negras sin saber como toman decisiones.
- **Etiquetado:** La satisfacción del usuario con las recomendaciones puede verse influenciada por el etiquetado de las recomendaciones, es común que en las búsquedas de Google aparezcan como primeros los ítems que están patrocinados con la etiqueta «Ad», es decir, le pagan a Google para aparecer más arriba en las y salen con esa etiqueta, algunos usuarios se saltan esos ítems y nunca los consumen, es por eso que las etiquetas pueden sesgar a los usuarios que consumen las recomendaciones.

2.2. Machine Learning

El Machine Learning o Aprendizaje Automático es una rama de la inteligencia Artificial y se centra en desarrollar que las computadoras **Aprendan** con la experiencia, los datos y el paso de las iteraciones. El concepto de aprendizaje puede ser visto como la tarea de explorar un gran espacio de hipótesis, definido implícitamente por nuestras representaciones de los datos, hipótesis, etc. Y encontrar la mejor hipótesis que se adapte a nuestros datos [Mitchell, 1997]. El Machine Learning tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis de mercado para los diferentes sectores de actividad, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica. Existen 4 grandes grupos en los que se clasifica los tipos de algoritmos de Machine Learning [Murphy, 2012] y son:

- **Aprendizaje supervisado:** Los datos están etiquetados y el algoritmo produce una función que mapea las entradas y las salidas deseadas del sistema
- **Aprendizaje no supervisado:** los datos no están etiquetados y se busca formar patrones para diferenciar y etiquetar nuevas entradas.
- **Aprendizaje semi-supervisado:** es una mezcla entre aprendizaje supervisado y no supervisado.
- **aprendizaje por refuerzo:** Existe un agente que interactúa con un ambiente mediante acciones las cuales otorgan distintas recompensas, la función de este aprendizaje es determinar que acciones debe escoger el agente para maximizar alguna función en base a las recompensas.

Para el caso de los sistemas de recomendación se podría decir que pertenecen al grupo del aprendizaje semi-supervisado, ya que si bien tenemos algunos datos con sus respectivas eti-

quetas, no tenemos las etiquetas para el total de los datos y nuestro objetivo es poder predecir de la manera más precisa esas etiquetas faltantes a futuro.

Además de la clasificación según aprendizaje en Machine Learning, también existe otra clasificación y es según la salida deseada de los modelos, existen 2 grandes grupos:

- **Regresiones:** Decimos que estamos frente a un problema de regresión cuando queremos predecir valores continuos, como por ejemplo predecir el precio de una casa, la temperatura, etc.
- **Clasificaciones:** En cambio, nos referimos a un problema de clasificación cuando queremos predecir y asignar clases a los datos, estos valores son discretos, como por ejemplo predecir qué animal aparece en una foto, si va a llover, estar nublado o soleado determinado día, etc.

En nuestro problema de sistemas recomendadores se utiliza un enfoque híbrido, ya que vamos a realizar regresiones para predecir numéricamente la calificación que un usuario asigna a un ítem, y también vamos a clasificar los ítems relevantes de los que no son relevantes para el usuario.

Por último, existen variadas técnicas de resolver estos problemas, como lo pueden ser:

- Árboles de Decisión
- Support Vector Machines (SVM)
- Redes Bayesianas
- Ensamblados
- Redes Neuronales

Por citar algunas técnicas de ejemplo, existen más, pero no se detallaran más a fondo cada uno, ya que nos estaríamos saliendo del tema principal por lo que se profundizará solamente la técnica más utilizada hoy en día para resolver el problema de los sistemas recomendadores que son las **Redes Neuronales**

2.2.1. Embeddings

Un embedding es una representación de baja dimensionalidad en el cual se pueden representar vectores de mayor dimensión. Esto resulta muy útil en los sistemas recomendadores, ya que el corpus de ítems a recomendar es muy grande y no es eficiente trabajar con datos altamente dimensionales, por lo que generamos embeddings o representaciones de los datos en un espacio con menor dimensión para poder trabajar de mejor manera, esto facilita el

poder realizar machine learning en entadas largas como vectores dispersos (Sparse Vectors) representando palabras. En el caso de NLP (Natural Language Processing) los embeddings capturan la semántica de las palabras y coloca muy juntos los vectores de los inputs similares en el espacio generado por el embedding. En palabras simples es llevar el texto a una representación numérica con la que se puedan realizar operaciones matemáticas, el modelo es el que se encarga de aprender estas representaciones o Embeddings.

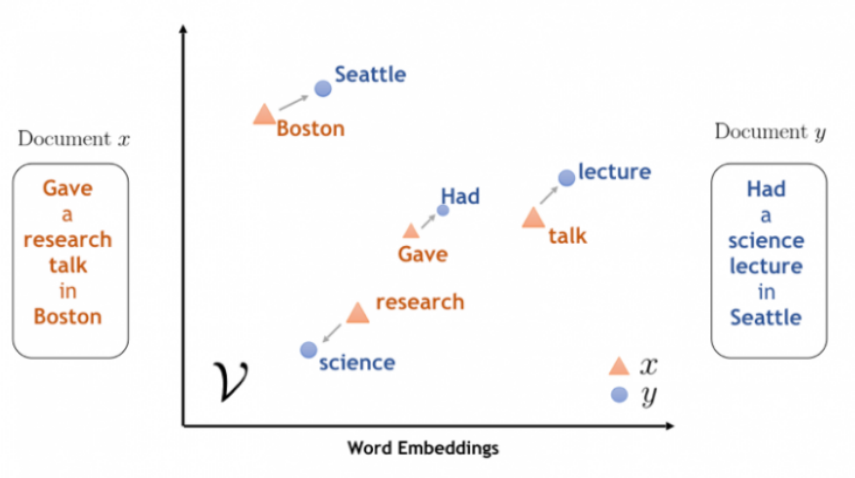


Figura 7: Word Embedding.
Fuente: IBM Research Blog.

En los modelos de redes neuronales, el objetivo es encontrar un **embedding** donde tanto usuarios como ítems puedan mapearse en conjunto, la predicción de ratings podría realizarse de esta manera:

$$r_{ui} = q_i^T \cdot p_u.$$

donde:

- q_i corresponde al vector de embedding del ítem i.
- p_u corresponde al vector de embedding del usuario u.

Para encontrar los valores de los vectores q_i y p_u se suele utilizar dos opciones

1. Opción 1 SVD: Usando **SVD**, que es una técnica de factorización matricial utilizada frecuentemente en recuperación de información, cosa en la cual no se profundizara en esta memoria porque no es la técnica utilizada.
2. Opción 2 Función de perdida: Usar una función de perdida, como la que planteamos en la sección 2.1.3, como un problema de optimización con regularización, que podemos

resolver con SGD u otras técnicas

$$\min_{q*,p*} \sum_{(u,i)\in K} (r_{ui} - q_i^T \cdot p_u)^2 + \lambda(\|q_i\|^2 + \|p_u\|^2).$$

donde K es el conjunto de pares (u, i) para los cuales r_{ui} es conocido (training set).

2.2.2. Medidas de similitud

Una medida de similitud es una función

$$s : E \times E \to \mathbb{R}.$$

que toma un par de embeddings y retorna un escalar midiendo su similitud. Los embeddings pueden ser usados para generar candidatos de la siguiente manera:

- dado una query $q \in E$, el sistema busca embeddings de ítems $x \in E$ que estén cerca de q, eso quiere decir que busca embeddings con alta similitud $s(q, x)$.

Para determinar el grado de similitud, la mayoría de los sistemas de recomendación utilizan:

- **Coseno:** es el coseno del ángulo entre dos vectores

$$s(q, x) = \cos(\overrightarrow{q}, \overrightarrow{x}) = \frac{\overrightarrow{q} \cdot \overrightarrow{x}}{\|\overrightarrow{q}\|_2 \times \|\overrightarrow{x}\|_2}.$$

- **Producto Punto:** el producto punto entre dos vectores se puede calcular de varias maneras, una de ellas es

$$s(q, x) = \langle q, x \rangle = \sum_{i=1}^{d} q_i x_i.$$

  También puede ser calculado como

$$s(q, x) = \|x\|\|q\| \cos(q, x).$$

  (el coseno del ángulo multiplicado por el producto de las normas). Por lo tanto, si los embeddings están normalizados, entonces el producto punto y el coseno coinciden.

- **Distancia Euclidiana:** es la distancia que se utiliza en un espacio euclidiano,

$$s(q, x) = \|q - x\| = [\sum_{i=1}^{d} (q_i - p_i)^2]^{\frac{1}{2}}.$$

Una distancia más pequeña significa una mayor similitud. Al igual que en el caso anterior, cuando los embeddings están normalizados, la distancia euclidiana al cuadrado coincide con el producto punto y el coseno a una constante tal que:

$$\frac{1}{2}\|q - x\|^2 = 1 - \langle q, x \rangle.$$

Dependiendo de la función de similitud que utilizamos obtendremos distintos resultados según sea el caso, en la figura siguiente tenemos el vector negro representando la query, y tres otros vectores de embeddings representando los distintos ítems, dependiendo de la función que utilizamos para medir la similitud los rankings de los ítems serán distintos.

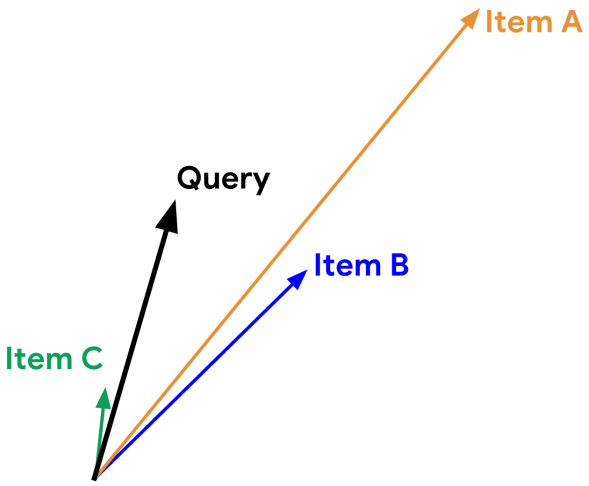


Figura 8: Vectores de embedding y Query.
Fuente: Google Developers.

El ítem A tiene la norma las larga y tiene mayor rating según el producto punto.

El ítem C tiene el ángulo más pequeño con respecto a la query y tiene mayor rating según la similitud del Coseno.

El ítem B es físicamente más cercano a la query, por lo tanto, tiene mayor rating según la distancia euclidiana.

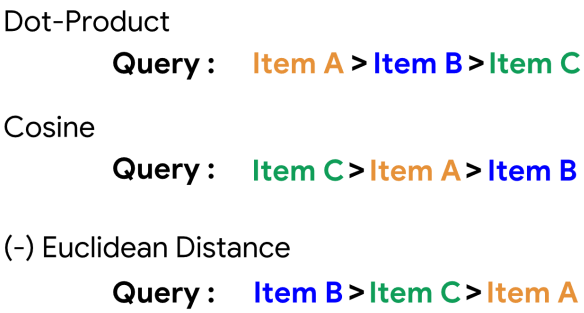


Figura 9: Orden de Ranking según similitud.
Fuente: Google Developers.

Comparado con el coseno, el producto punto es sensitivo a la norma de los embeddings, eso significa que mientras más grande sea la norma de un embedding, mayor será su similitud (para ítems con ángulos agudos), esto puede afectar a las recomendaciones de la siguiente manera:

- ítems que aparecen frecuentemente en el training set (vídeos populares en YouTube o canciones populares en Spotify), tienden a tener embeddings con normas grandes. Si deseamos capturar la información de popularidad de los ítems, entonces es preferible utilizar el producto punto. Por otra parte, si no somos cuidadosos, los ítems populares terminarán dominando las recomendaciones.
- ítems que son muy poco frecuentes, puede que sus embeddings no sean actualizados frecuentemente durante el training set, por lo que si se inicializan con normas grandes, el sistema comenzará a recomendar ítems raros por sobre los ítems relevantes, es por esto que hay que tener cuidado a la hora de inicializar los embeddings y utilizar métodos de regularización apropiados.

2.3. Redes Neuronales

Las redes neuronales son un subcampo del Machine Learning (Ver figura 10) que ha ganado muchísima relevancia estos últimos años, ya que poseen muchísima capacidad de generalización, permitiéndoles resolver problemas muy distintos y con las mejores soluciones, la mayoría de estados del arte en problemas de inteligencia artificial están hechos con redes neuronales.

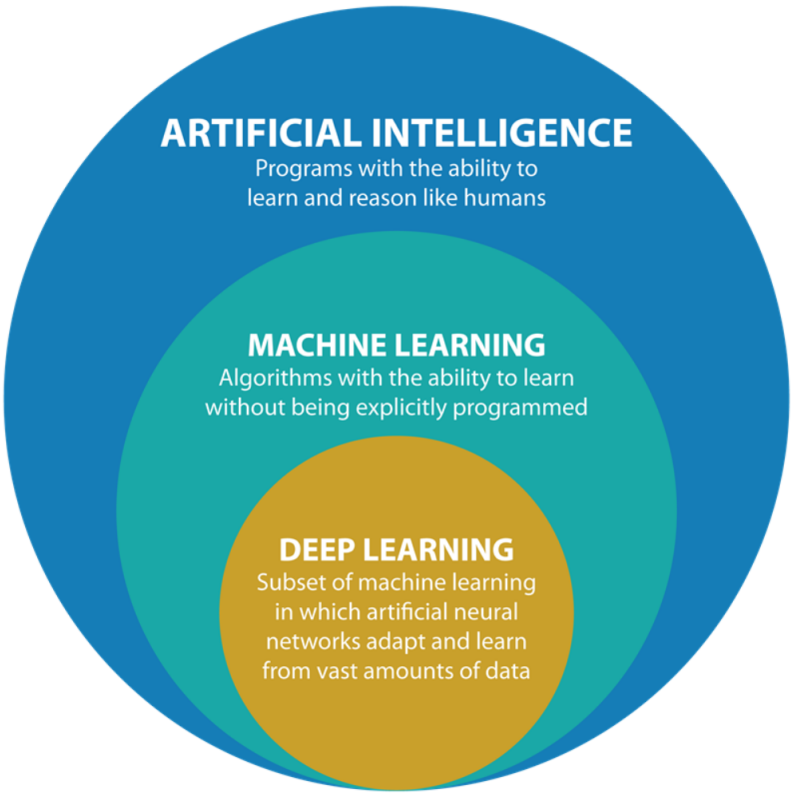


Figura 10: Agrupamientos de la IA, ML y DL.
Fuente: Zach Johnson, ZekiahTech.

Una red neuronal son un grupo de neuronas o nodos agrupados por capas por donde fluye la información, toda esta idea proviene de la biología y es que las redes neuronales están basadas en el funcionamiento del cerebro y sus neuronas. Cada una de estas neuronas poseen una función de activación y están unidas a pesos que conectan las salidas con las entradas de otras neuronas, así lo que se busca es que la red aprenda y modifique estos pesos que conectan neuronas entre capas para así minimizar la función de perdida, una métrica que utiliza el modelo para evaluar las predicciones realizadas por la red y los valores reales [Goodfellow *et al.*, 2016].

2.3.1. Neuronas

Una neurona es un nodo por donde fluye información, este nodo está basado en las neuronas biológicas del cerebro (Ver figura 11). Las entradas de estas neuronas son las salidas de las neuronas de la capa anterior multiplicadas por los pesos respectivos, todos estos valores pasan a través de una función de activación que se transforma los resultados a un espacio no lineal para poder modelar patrones complejos.

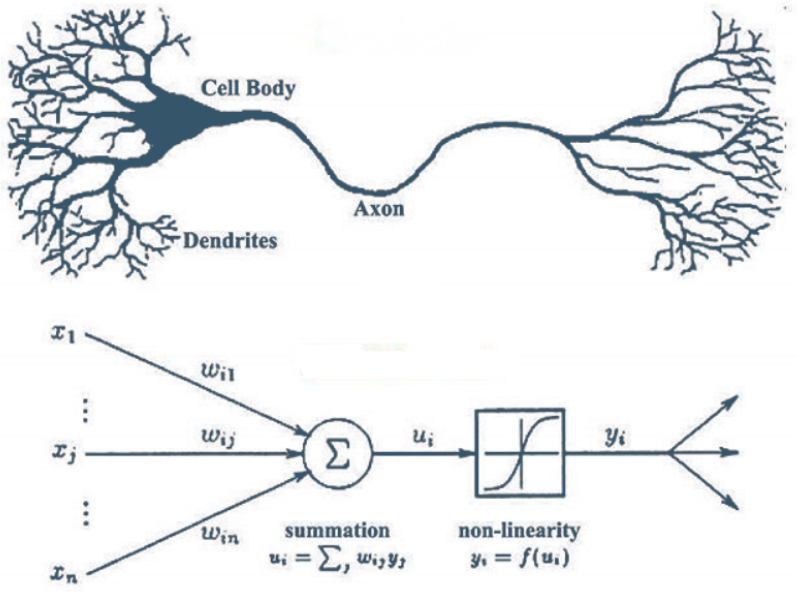


Figura 11: Neurona biológica y Neurona Artificial.
Fuente: Systems, Magiquo.

En términos computacionales y matemáticos, las neuronas no son más que muchas sumas y multiplicaciones a las cuales generalmente se les aplica una función no lineal.

$$z = f(b + x \cdot w) = f(b + \sum_{i=1}^{n} x_i w_i).$$

También podemos representar las neuronas en forma de vectores y matrices, permitiéndonos poder paralelizar los cálculos, por esta razón es que se utilizan GPU (Graphics Processing Units) o TPU (Tensor Processor Units) [Jouppi *et al.*, 2023], procesadores especializados para inteligencia artificial y machine learning, ambos chips se utilizan ampliamente para acelerar el entrenamiento de las redes neuronales.

$$\begin{bmatrix} z_{11} \\ z_{12} \\ z_{13} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \begin{bmatrix} x1 \\ x2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_1 \\ b_1 \end{bmatrix}$$

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

Figura 12: Fórmula de una neurona en forma Matricial.
Fuente: Study Machine Learning.

Donde **W** son los pesos que acompañan a cada feature de la red, **B** que es el bias o él intercepto de una ecuación linear, y por último **X** que es el vector con las features o características de nuestros datos.

2.3.2. Capas

Las redes neuronales se organizan por capas, existen tres grandes grupos de capas:

- **Input Layer:** es la capa de entrada donde se reciben los datos con los que se va a trabajar, ya sean datos tabulares, imágenes, series de tiempo, etc.
- **Hidden Layer:** son las capas intermedias entre la input layer y la output layer, en estas capas el modelo va aprendiendo patrones mediante la representación de variables latentes, a mayor cantidad de hidden layers o a mayor profundidad se dirá que el modelo es *Deep*, por lo que es muy común referirse a modelos de redes neuronales profundos como **Deep Learning**.
- **Output layer:** es la última capa del modelo y donde se producirán los resultados, después de esta capa se calculara el error mediante la función de perdida escogida, para problemas de regresión se suele utilizar neuronas con función de activación ReLU o Lineales, y para problemas de clasificación se suele utilizar Softmax como función de activación, en casos de clasificación binaria también se puede utilizar la función Sigmoide.

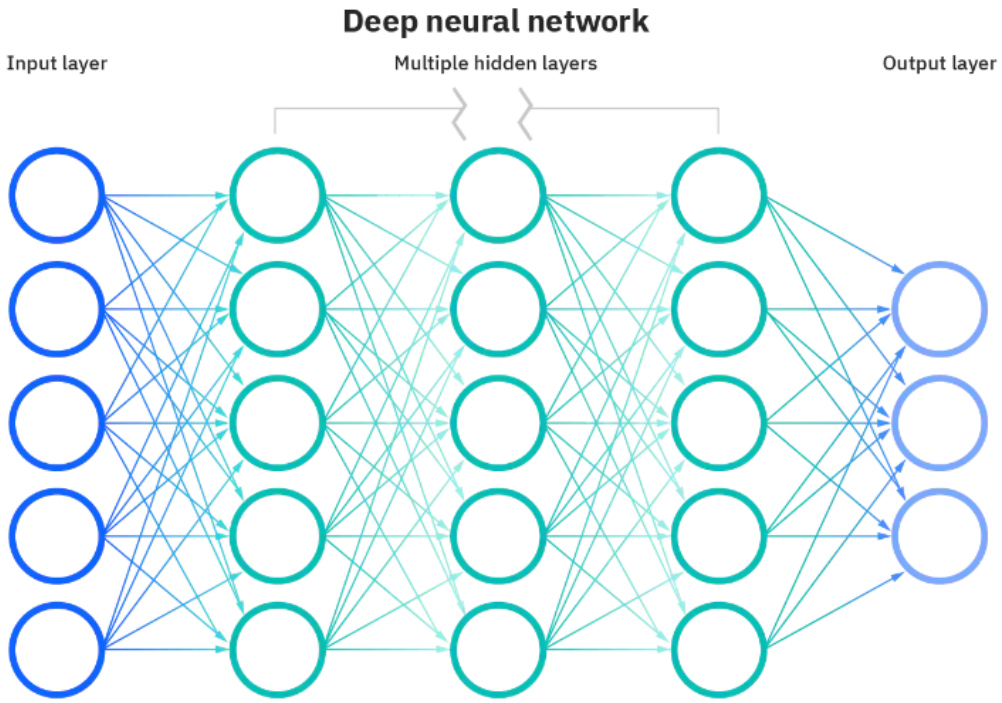


Figura 13: Capas de una red neuronal.
Fuente: Cloud Education, IBM.

Pero esta no es la única clasificación que se le puede dar a las capas de una red neuronal, también se clasifican según la función que cumplen, algunas capas son:

- Capas de activación
- Capas Convolucionales
- Capas de Regularización
- Capas de Normalización
- Capas Recurrentes
- Capas de Preprocesamiento

2.3.3. Métodos de inicialización de pesos

La inicialización de los pesos de una red neuronal es una parte importante para el buen desempeño de la misma, existen variadas formas de inicializar los pesos de una red, algunas técnicas son:

- Zeros
- Random
- Xavier/Glorot Initialization
- Kaiming/He Initialization
- T-Fixup

Xavier Glorot
La inicialización de Xavier Glorot es uno de los métodos más utilizados para inicializar los pesos de una red neuronal [Glorot y Bengio, 2010], los biases se inician en 0 y los pesos W_{ij} de cada capa son inicializados como:

$$W_{ij} \sim U[-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}].$$

Donde U es una distribución uniforme y fan_{in} es el tamaño de la capa anterior (Número de columnas en W) y fan_{out} es el tamaño de la capa actual.

Kaiming He

La inicialización de He o Kaiming, surge como solución para inicializar las capas con funciones del tipo ReLU [He *et al.*, 2015b], ya que la inicialización de Glorot está pensada para ser utilizada con capas con funciones Sigmoides o Tanh. En este método los biases se inician en 0 y los pesos W_{ij} de cada capa son inicializados como:

$$W_l \sim \mathcal{N}\left(0, \frac{2}{n_l}\right).$$

Eso es una distribución Gaussiana centrada en cero y con una desviación estándar de $\sqrt{2/n_l}$, con n_l siendo el número de neuronas por cada capa l.

2.3.4. Regularizadores

Para que el modelo no sufra de **Sobre Ajuste** (OverFitting) se aplican distintas técnicas de regularización que penalizan distintos aspectos de la red, ya que fácilmente podemos aumentar la complejidad del modelo y permitirle que aprenda cualquier representación, pero ya no estaría generalizando que es lo que buscamos, sino que estaría memorizando y cuando se enfrente a datos que no haya visto antes, va a fallar, por lo que con la regularización se busca penalizar a la red para que no memorice y generalice lo mejor posible. Por otro lado, si aplicamos mucha regularización podemos hacer que la red sea incapaz de modelar correctamente el problema, sufriendo del efecto contrario al sobre ajuste que sería el **UnderFitting**

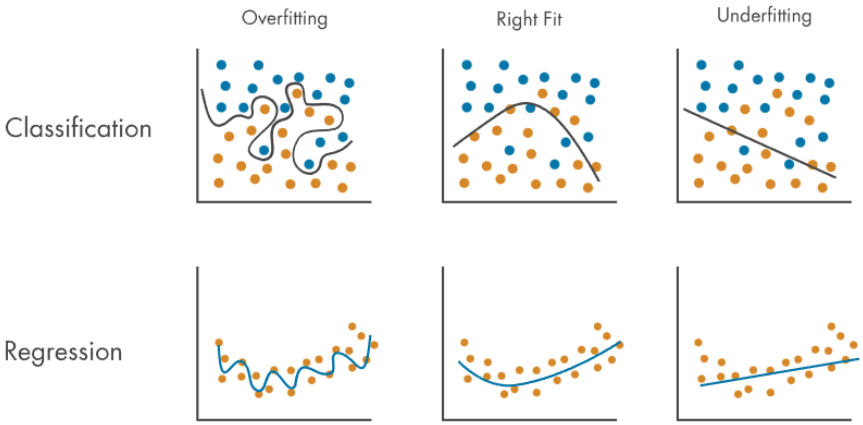


Figura 14: Overfitting y UnderFitting.
Fuente: MathWorks, Matlab.

Existen varias formas de aplicar regularización (L1, L2, L1L2), ya sea en los pesos de la red, en los biases, en la capa completa, como también existen capas especializadas en solo aplicar métodos de regularización, como lo es el **Dropout** [Srivastava *et al.*, 2014].

Dropout
Dropout es una técnica de regularización que consiste en apagar u omitir neuronas de una capa con una probabilidad p en la fase de entrenamiento, en la fase de test todas las neuronas están presentes, pero los pesos son escalados por p.

La idea es prevenir la co-adaptación, que es cuando la red neuronal se vuelve demasiado dependiente en ciertas conexiones, ya que esto podría ser síntoma de *OverFitting*, de esta manera aplicamos una máscara que apaga neuronas dependiendo una probabilidad y así en cada batch de entrenamiento distintas neuronas están activas y otras apagadas, forzando a la red a no solo aprender un camino de conexiones para generar la predicción, sino que aprende múltiples caminos y conexiones para generalizar de mejor manera.

Esta técnica ha demostrado ser muy útil y es un estándar hoy en día en los modelos profundos de redes neuronales, además se han propuesto distintas versiones de dropout tales como:

- Spatial Dropout
- Gaussian Dropout
- Alpha Dropout

2.3.5. Funciones de Activación

Las funciones activación son funciones matemáticas que transforman el espacio, esto se realiza para modelar patrones más complejos, algunas veces nos sirven para decidir si la neurona estará o no activa, ya que hay ciertas funciones de activación como la sigmoide que transforman los resultados a un espacio [0,1]. Algunas de las funciones de activación son:

- **ReLU** (Rectified linear unit): $f(x) = max(0, x)$.
- **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$.
- **Tanh** (Tangente Hiperbólica): $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$.
- **Softmax**: $\frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \forall i \in [1, ..., J]$.
- **Softplus**: $f(x) = \log(1 + e^{x})$.

Cada una de estas funciones y las muchas más que existen tienen sus ventajas y desventajas, por lo que se utilizan acorde al problema que se busca solucionar.

GELU
Gaussian Error Linear Unit o GELU [Hendrycks y Gimpel, 2020], es una función de activación que pondera los pesos por su percentil en lugar de su signo como lo haría ReLU, en consecuencia GELU se considera una función más suave que ReLU

$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)].$$

Donde *erf* es la función de error o **Gauss error Function** y $X \sim \mathcal{N}(0, 1)$. La función GELU es utilizada ampliamente en arquitecturas de Transformers como lo son GPT-3 y BERT.

2.3.6. Forward Pass

Forward pass hace referencia al proceso donde se calculan todos los valores para las neuronas partiendo desde la capa de entrada y terminando en la capa de salida, es un proceso bastante simple y lineal donde se van multiplicando los valores de salida de las neuronas por los respectivos pesos.

2.3.7. Backpropagation

Para que la red logre ajustar los pesos en base a los errores y así poder aprender, se utiliza Backpropagation, este es el proceso donde a partir del error de la capa de salida, vamos retransmitiendo el error hacia atrás a las distintas capas y ajustando los pesos mediante el cálculo del gradiente de la función de perdida con respecto a los distintos pesos de la red, como cada peso depende de los valores de las neuronas anteriores se utiliza la regla de la cadena para calcular estas derivadas parciales. Para poder ajustar los pesos de manera adecuada sin hacerlo de manera tan agresiva o tan despacio, se utiliza un hiperparámetro llamado **tasa de aprendizaje** (Learning Rate), con el uso de optimizadores más complejos, este learning rate se va ajustando con el paso de las iteraciones.

2.3.8. Learning Rate

El learning rate es un hiperparámetro crítico en algoritmos de optimización utilizados en el entrenamiento de modelos de aprendizaje automático, como redes neuronales y algoritmos de descenso de gradiente. Representa la magnitud de los pasos que el algoritmo toma en la dirección opuesta al gradiente de la función de pérdida durante el proceso de optimización. Determina cuánto se ajustan los pesos del modelo en cada iteración durante el proceso de entrenamiento. Si el learning rate es muy pequeño, el entrenamiento puede ser lento y requerir muchas iteraciones para converger. Por otro lado, si el learning rate es muy grande, el entrenamiento puede volverse inestable y la función de pérdida puede oscilar o incluso divergir.

Scheduled Learning Rate
Scheduled learning rate (tasa de aprendizaje programada) es una técnica en la que se ajusta el valor del learning rate durante el entrenamiento de un modelo de aprendizaje automático en función de cierto patrón o programación predefinida. En lugar de mantener el mismo valor constante para el learning rate durante todo el proceso de entrenamiento, se modifican sus valores en momentos específicos o según ciertas condiciones.

Warm-Up Learning Rate
Él *warm-up* es una manera de reducir el efecto de sobreajuste en los ejemplos de entrenamiento tempranos. Si el dataset de entrenamiento está altamente diferenciado, el entrenamiento inicial del modelo podría sesgarse hacia esas características, la técnica de warm-up implica comenzar con una tasa de aprendizaje pequeña y luego aumentar gradualmente su valor a medida que avanza el entrenamiento.

2.3.9. Optimizadores

Los optimizadores son los algoritmos que van modificando los atributos de la red neuronal como los pesos y el learning rate, existen distintos tipos de optimizadores y se escogen basándose en la tarea que la red tiene que resolver. La mayoría de los optimizadores se basan en el **descenso del gradiente** (Gradient Descent), que es ir poco a poco ajustando los pesos dependiendo del learning rate en función de la dirección opuesta del gradiente para encontrar un óptimo local (Ver Figura 15).

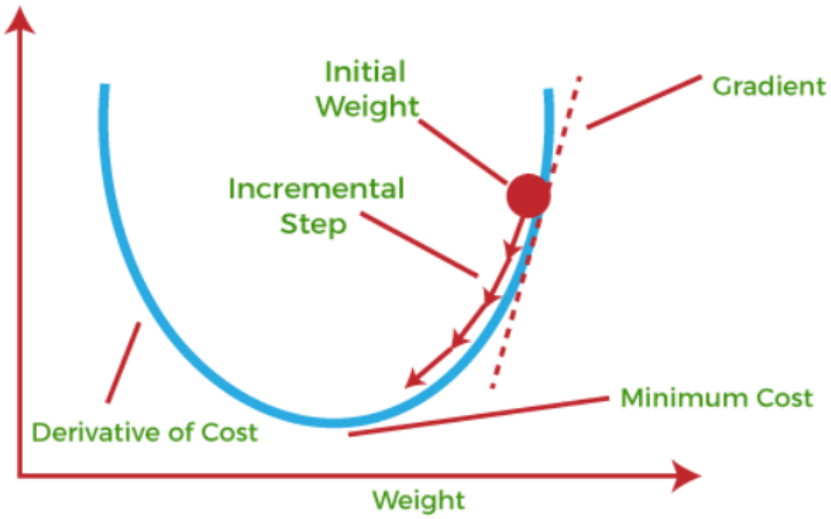


Figura 15: Gradiente Descendente.
Fuente: JavaTPoint.

Existen distintos tipos de optimizadores y debido a que se escapa de los márgenes de esta memoria, no se procederá a explicar cada uno en detalle, más adelante se explicara en detalle solo el optimizador que se utilizó para este problema, pero los más comunes son:

- SGD (Stochastic gradient descent)
- AdaGrad
- RMSProp
- Adam
- AdamW

AdamW
AdamW [Loshchilov y Hutter, 2019], es una versión modificada de Adam que busca resolver el problema de regularización L2 y Weight Decay en optimizadores con métodos de gradientes adaptativos (AdaGrad, RMSProp, Adam, Etc.).

Ilya Loshchilov y Frank Hutter proponen AdamW, esto basándose en que la regularización L2 con gradientes adaptativos no es equivalente a weight decay, ya que al comparar Adam con SGD, los pesos de los gradientes se regularizan menos de lo que deberán si se utilizara SGD con weight decay. Como se puede observar la implementación de la regularización L2 en Adam es:

$$g_t = \nabla f(\theta_t) + w_t \theta_t.$$

donde w_t es el porcentaje de cambio del weight decay en un tiempo t.
En cambio, AdamW ajusta el término weight decay para que aparezca en la actualización del gradiente

$$\theta_{t+1,i} = \theta_{t,i} - \eta(\frac{1}{\sqrt{\hat{v}_t + \epsilon}} \cdot \hat{m}_t + w_{t,i}\theta_{t,i}), \forall t.$$

Este pequeño cambio aplica adecuadamente weight decay en Adam.

AMSGrad
AMSGrad [Reddi *et al.*, 2018], es un método de optimización estocástico que busca arreglar y mejorar la convergencia con los optimizadores basados en Adam, AMSgrad utiliza el máximo de los gradientes pasados v_t en vez de la media exponencial para actualizar los parámetros. Generalmente, Adam adapta automáticamente un learning rate separado para cada parámetro en el problema de optimización, esto genera una limitación, ya que si bien es bueno que Adam disminuya el learning rate cuando se acerca al óptimo, esto puede generar que se aumenten las iteraciones necesarias para llegar a ese óptimo lo cual es malo. AMSGrad es una extensión de Adam que mantiene un máximo del vector del segundo momento y lo

utiliza para corregir el bias del gradiente para actualizar los parámetros, en vez de utilizar el vector del momento en sí mismo, esto ayuda a prevenir la **convergencia prematura** (Premature Convergence) que ralentiza la optimización.

$$
\begin{aligned}
m_t &= \beta_1 m_{t_1} + (1-\beta_1) g_t, \\
v_t &= \beta_2 v_{t_1} + (1-\beta_2) g_t^2, \\
\hat{v_t} &= max(v_{t-1}, v_t), \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t.
\end{aligned}
$$

2.4. Transformers y Atención

En las redes neuronales, la **Atención** es una técnica que trata de imitar la *Atención Cognitiva* que utilizamos los humanos en el día a día, ya que no le damos la misma importancia a todas las cosas que percibimos, con nuestra atención podemos darle mayor o menor importancia a las cosas dependiendo el contexto, por ejemplo cuando vemos una película en un cine, utilizamos nuestra atención para fijarnos en la pantalla y asignarle mucha importancia, en cambio, ignoramos o no nos percatamos de las cosas que suceden al rededor en la sala de cine como pueden ser las otras personas dentro de la sala, por lo que les damos poca importancia, esto mismo es lo que buscan los mecanismos de atención, dependiendo el contexto poder asignarle distintos grados de importancia para que así el modelo pueda desempeñarse de mejor manera.

Los mecanismos de atención fueron introducidos en la década de 1990, bajo nombres como módulos multiplicativos, hyper-networks, etc.

2.4.1. Neural Machine Translation

El problema que se buscaba resolver era poder traducir largas secuencias de texto en *Neural Machine Translation* (NMT). En problemas de traducción lo que se solía utilizar eran redes recurrentes con una arquitectura encoder-decoder, lo difícil de traducir estas largas secuencias es que los gradientes y la información se va desvaneciendo en las redes recurrentes a medida que la secuencia es más larga y además la información del texto original solo la capturaba la primera capa del decoder, por lo que en 2016 *Yoshua Bengio, KyungHyun Cho y Dzmitry Bahdanau* publican una investigación [Bahdanau *et al.*, 2016] proponiendo un método de alineación con un **vector de contexto** para darle mayor información a las redes recurrentes bidireccionales.

El vector de contexto depende de una secuencia de anotaciones h que el encoder mapea con los inputs, cada anotación h_i contiene información de toda la secuencia con un fuerte foco

en las partes cercanas a la i-ésima palabra de la secuencia de entrada. El vector de contexto c_i es computado como una suma ponderada de esas anotaciones, h_i

$$c_i = \sum_{j=1}^{T_s} \alpha_{ij} h_j.$$

donde el peso α_{ij} de cada anotación h_j es computado por:

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})}.$$

donde:

$$e_{ij} = a(s_{i-1}, h_j).$$

es un modelo de alineación que puntúa cuan bien calzan los inputs al rededor de la posición j y el output en la posición i. por último, la puntuación es parametrizada vía una red neuronal feedforward a con una hidden layer que se entrena en conjunto con los otros componentes del sistema. La función de puntuación de alineación sigue la siguiente forma dado que se utiliza *tanh* como función de activación no lineal,

$$score(s_t, h_i) = v_a^T tanh(W_a[s_t, h_i]).$$

donde ambas v_a y W_a son matrices de pesos que el modelo de alineación aprende con los datos. La matriz de puntajes de alineación es buen subproducto para mostrar explícitamente la correlación entre las palabras de input y de output.

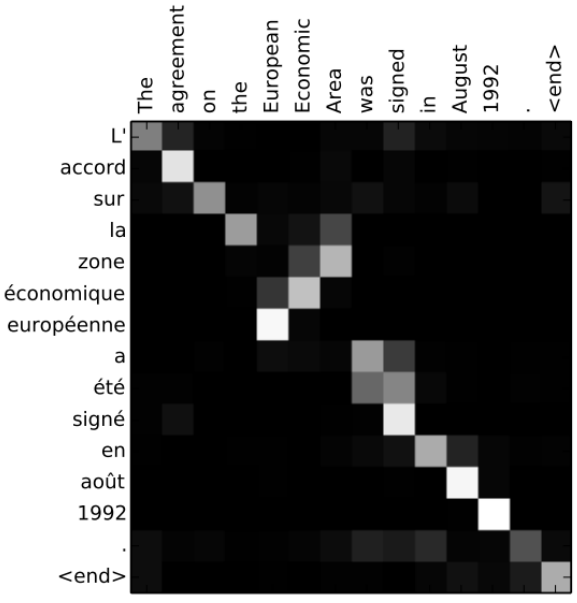


Figura 16: Matriz de Alineación al traducir del francés al inglés.
Fuente: [Bahdanau *et al.*, 2016], Neural Machine Translation by Jointly Learning to Align and Translate.

2.4.2. Transformers

Un año después, investigadores de Google publican su propia arquitectura especializada para resolver el problema de **NMT**, proponiendo el **Transformer** en su investigación *Attention Is All You Need* [Vaswani *et al.*, 2017]. La arquitectura del transformer también tiene una estructura encoder-decoder, y utiliza capas apiladas de auto-atención (self-attention) y feedforwards fully-conected tanto para el encoder como para el decoder.

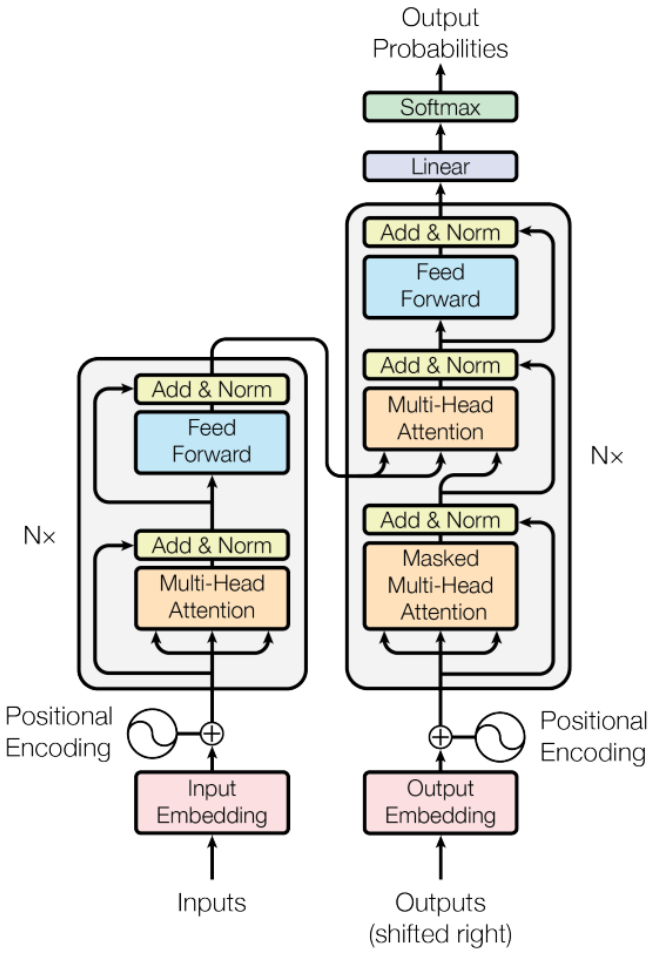


Figura 17: Arquitectura Transformer.
Fuente: [Vaswani *et al.*, 2017], Attention Is All You Need.

Como se puede observar en la imagen anterior, el transformer tiene varios componentes, pero solo nos centraremos en los más importantes para el problema de sistemas recomendadores.

Encoder
Cada Capa del encoder tiene dos sub capas, la primera es un mecanismo de auto-atención multi-cabezal y la segunda es una feedforward fully connected con una hidden layer, se utilizan **conexiones residuales** [He *et al.*, 2015a] (Capas que *saltan* transformaciones permitiendo preservar información) y **capas de normalización** [Ba *et al.*, 2016] (todas las neuronas en una capa particular tienen la misma distribución).

Decoder
Además de las dos sub capas del encoder, el decoder inserta una tercera sub capa que ejecuta atención multi-cabezal sobre las salidas del enconder, también se modifica la auto-atención del decoder con una máscara, esta máscara junto con que los outputs se mueven una determinada posición para asegurar que las predicciones de la posición i solo dependan de las salidas conocidas previas a i.

Scaled Dot-Product Attention
la entrada consiste en **queries** y **keys** de dimension d_k y **Values** de dimensión d_v. se computa el producto punto entre las queries y las keys dividiendo cada una por $\sqrt{d_k}$ y aplicando una softmax para obtener los pesos de los values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V.$$

Donde Q, K, V son matrices de queries, keys y values respectivamente. La atención utilizada es del tipo producto punto (dot-product) y consiste en la suma de las componentes de dos vectores, geométricamente se puede ver como el producto entre las magnitudes euclidianas de dos vectores y el coseno del ángulo entre ellos

$$\text{Algebraicamente: } a \cdot b = \sum_{i=1}^{n} a_n b_n.$$

$$\text{Geométricamente: } a \cdot b = \|a\|\|b\| \cos\theta.$$

$$\text{donde: } \cos\theta = \frac{a \cdot b}{\|a\|\|b\|}.$$

Si lo vemos de forma algebraica, esto nos indica el grado de similitud entre dos vectores en un espacio n-dimensional, que efectivamente es lo que buscamos con los embeddings de usuarios y productos en un sistema recomendador.

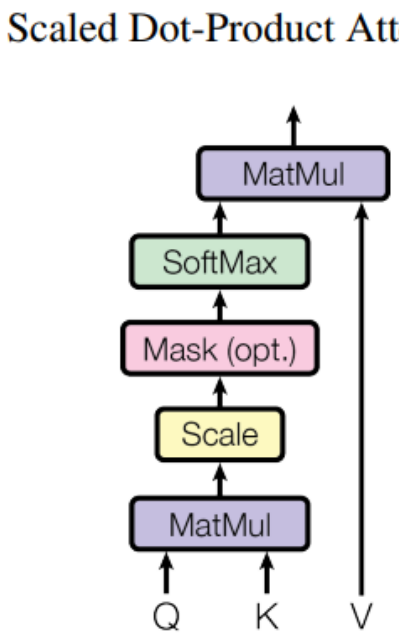


Figura 18: Scaled Dot-Product Attention.
Fuente: [Vaswani *et al.*, 2017], Attention Is All You Need.

Multi-Head Attention
La atención multi-cabezal (Multi-Head Attention) consiste en computar en paralelo múltiples **Scaled Dot-Product Attention** proyectadas linealmente h veces con diferentes proyecciones aprendidas por la red a d_k, d_k, d_d dimensiones para las queries, keys y values respectivamente, luego concatenar las proyecciones y proyectar el resultado linealmente una última vez.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O.$$

Donde:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V).$$

en la cual las proyecciones son las siguientes matrices:

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}.$$

Y por último:

$$W^O \in \mathbb{R}^{hd_v \times d_{model}}.$$

Donde d_{model} es la dimensión de los embeddings.

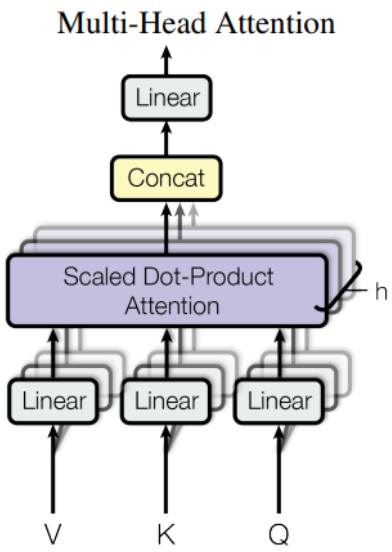


Figura 19: Multi-Head Attention.
Fuente: [Vaswani *et al.*, 2017], Attention Is All You Need.

Self-Attention

En una capa de auto-atención (Self-Attention), todas las keys, values y queries provienen del mismo lugar, esto se aplica para el caso del encoder y la primera sub-capa del decoder, ya que en la segunda sub-capa del decoder las queries y keys provienen del enconder, es importante remarcar esta diferencia, ya que la auto-atención lo que busca es la similitud entre los propios datos de entrada, en un sistema recomendador lo que necesitamos son 2 bloques de enconder, uno para cada torre de Queries y Candidatos respectivamente, que contengan auto-atención entre sus embeddings, esto nos permite analizar la relación usuario con usuario entre todos los usuarios y producto con producto entre todos los productos, para así después calcular una atención general entre usuarios y productos.

	Variable 1	*Variable 2*	*Variable 3*	*Variable 4*	*Variable 5*	*Variable 6*	*Variable 7*	*Variable 8*	*Variable 9*	*Variable 10*
Variable 1	1									
Variable 2	0.272724	1								
Variable 3	-0.03205	0.267882	1							
Variable 4	0.108167	0.282454	0.344223	1						
Variable 5	-0.03914	-0.36504	-0.01383	-0.37275	1					
Variable 6	0.013927	-0.36842	-0.03052	-0.14839	-0.20021	1				
Variable 7	-0.08183	0.307122	0.596709	0.539091	-0.23986	0.001919	1			
Variable 8	0.270934	-0.17073	0.057761	-0.05328	-0.18467	0.311216	-0.02321	1		
Variable 9	0.213589	0.161635	0.156619	0.159255	-0.26331	0.059653	0.066852	0.18052	1	
Variable 10	-0.03829	-0.07516	-0.31795	-0.12114	-0.21915	-0.28395	-0.34985	-0.07299	-0.31734	1

Figura 20: Matriz de atención para 10 variables.
Fuente: Steven Bradburn, toptipbio.

Como se puede observar en la imagen anterior, esa matriz de atención de 10 variables perfectamente puede ser una matriz de atención de diez usuarios entre sí o diez productos entre sí, esto es lo que buscamos aplicando auto-atención en sistemas recomendadores.

2.4.3. Pre-LN Transformer

Existen múltiples formas de implementar modelos de atención, cada una con sus distintas variaciones del Transformer (Reformer, Linformer, Longformer, etc.) ([Lin *et al.*, 2021]), estos cambios pueden provenir desde cambios en la arquitectura, funciones de activación, el orden de las capas, sparsing en los métodos de atención, etc. Uno de los variantes más populares del transformer vanilla, es el Pre-LN Transformer ([Xiong *et al.*, 2020]), modelo propuesto para solucionar algunos problemas relacionados con la propagación del gradiente, el warm-up del learning rate, tuneo de hiperparámetros y el entrenamiento cuando los modelos son muy profundos.

Para entrenar un Transformer, generalmente se necesita una etapa cuidadosamente diseñada de calentamiento (Warm-up) del learning rate, que se ha demostrado que es crucial para el rendimiento final, pero ralentiza la optimización y requiere más ajustes de hiperparámetros. En el paper *On Layer Normalization in the Transformer Architecture* los autores proponen la siguiente fórmula para la etapa de warm-up del learning rate:

$$lr(t) = \frac{t}{T_{warmup}} lr_{max}, \quad \forall t \leq T_{warmup}.$$

Se denota la tasa de aprendizaje de la t-ésima iteración como $lr(t)$ y la tasa máxima de aprendizaje durante el entrenamiento como lr_{max} y un marco de tiempo predefinido T_{warmup}.

Después de esta etapa de warmup, el learning rate será establecido por schedulers de learning rate clásicos, como el Lineal decay, el inverse square-root decay, o exponetial decay, etc.

otro problema es que en el modelo Transformer original, la normalización se aplica después de cada capa de atención hacia adelante. Sin embargo, se ha observado que esta secuencia de normalización puede dificultar el flujo de gradiente a lo largo de la red y afectar negativamente al rendimiento del modelo en tareas más complejas.

Pre-LN Transformer aborda estos problemas reorganizando el orden de las operaciones de normalización y transformación en cada capa. En lugar de aplicar la normalización después de cada capa, se aplica antes. Esto significa que se aplica la normalización de capa (layer normalization) a la entrada de cada sub capa en lugar de aplicarla a la salida. Luego, se realiza la transformación de la sub capa y se agrega la conexión residual.

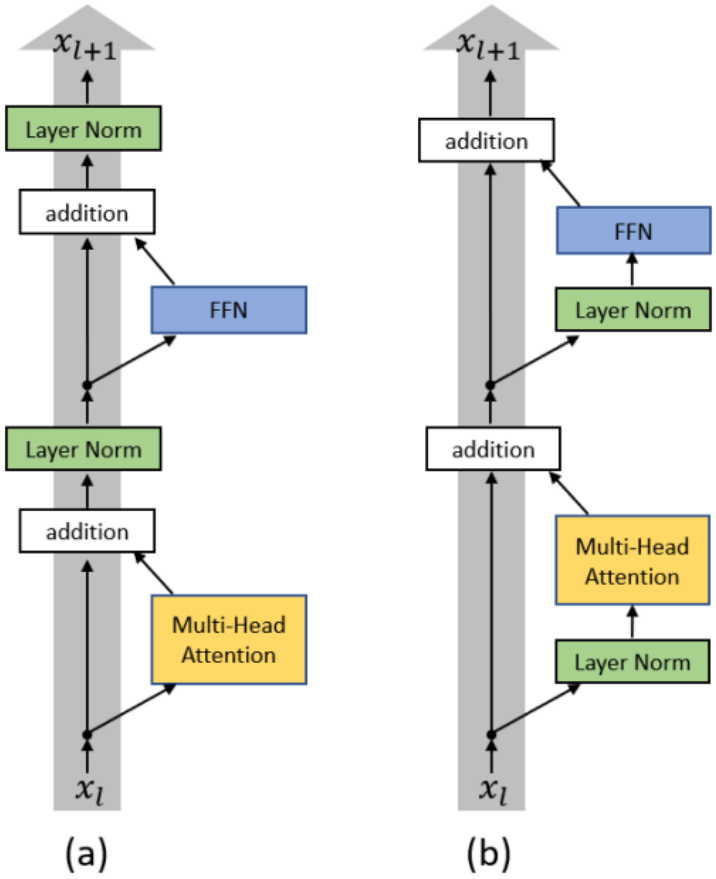


Figura 21: Comparación (a) Post-LN Transformer (Vanilla); (b) Pre-LN Transformer. Fuente: [Xiong *et al.*, 2020].

Esta reorganización permite que el gradiente fluya más fácilmente a través de la red, ya que la normalización previa ayuda a reducir la propagación de la varianza en los valores de activación. Además, al aplicar la normalización antes de la transformación, el modelo puede beneficiarse de una mejor regularización y una mayor estabilidad numérica durante el entrenamiento. Además, los autores demuestran como este cambio en el orden de las capas permite eliminar la etapa de Warm-up para el learning rate de manera segura, reduciendo el número de hiperparámetros de la red neuronal, además mencionan que la función de perdida decae mucho más rápido para el Pre-LN Transformer, lo que se traduce a que se pueden obtener resultados similares en menos iteraciones, lo que reduce el tiempo de entrenamiento, algo muy importante para tareas con datasets grandes y complejos.

Al ser un cambio tan pequeño como el orden de las capas, su implementación no fue costosa, es por esta razón que se incluyo una versión con el Pre-LN Transformer a los experimentos para ver como se comporta esta variante.

2.4.4. Aplicaciones

Esta arquitectura ha sido toda una revolución dentro del mundo del Deep Learning, y el Transformer es la base de grandes modelos de inteligencia artificial como lo es **ChatGPT**, un **Large Language Model** (LLM) que genera texto muy preciso gracias al Pre-entrenamiento Generativo [Radford *et al.*, 2018], actualmente está basado en GPT-4 (Generative Pretrained Transformer Version 4) [OpenAI, 2023]. Actualmente, no solo se aplica a texto, sino que se aplica en una variedad de áreas, como en problemas de Visión por computador (CV) [Dosovitskiy *et al.*, 2021] y Sistemas Recomendadores, estos últimos son en los que profundizaremos más, ya que la base de un sistema recomendador es poder emparejar de la mejor forma usuarios y artículos, objetivo que también se busca con la atención, es por esto que se aplicarán métodos de auto-atención en cada una de las torres (Query-Candidate) que componen a nuestro sistema recomendador.

Existen distintos enfoques a la hora de utilizar la arquitectura de transformers, ya que se puede utilizar solo la parte del decoder, o utilizar solo la parte del decoder. **BERT** (Bidirectional Encoder Representations from Transformers) [Devlin *et al.*, 2019] es un LLM compuesto de solo capas de encoders, y este modelo se utiliza ampliamente para generar embeddings ya entrenados y no necesitar entrenar desde 0 los embeddings para problemas de NLP.

2.5. Neural Recommender Systems

Los sistemas de recomendación que utilizan redes neuronales se les suele llamar **Neural Recommender Systems** o **Deep Recommender Systems**. Los sistemas de recomendación generalmente están compuestos por tres etapas o componentes (Ver figura 25):

2.5.1. Retrieval Stage (Candidate Generation)

En esta etapa el modelo se encarga de seleccionar un conjunto inicial de candidatos de entre todos los candidatos posibles. El objetivo principal de esta etapa es eliminar de manera eficiente a todos los candidatos sobre los cuales el usuario no tiene interés. Debido a que en esta etapa se puede tratar con millones de candidatos, esta etapa debe ser computacionalmente eficiente. El retrieval Stage está compuesto generalmente de dos modelos, Query y Candidate.

- **Query Model:** El modelo de Query computa la representación del contexto mediante los vectores de embeddings de los usuarios
- **Candidate Model:** El Candidate model computa la representación de los candidatos utilizando las features de los candidatos mediante los vectores de embeddings de los ítems.

Los resultados de los dos modelos luego se multiplican para dar una puntuación de afinidad entre la query y el candidato, una vez tengamos el embedding de la query q de los usuarios, podemos buscar los embeddings de los ítems V_j que se encuentran cercanos a q en el espacio de embeddings, esto se convierte en un problema de **nearest neighbor** [Jannach *et al.*, 2010] donde se retornan los top-k ítems de acuerdo al valor de similitud $sim(q, V_j)$, en el cual las puntuaciones más altas expresan una mejor coincidencia entre el candidato y la query.

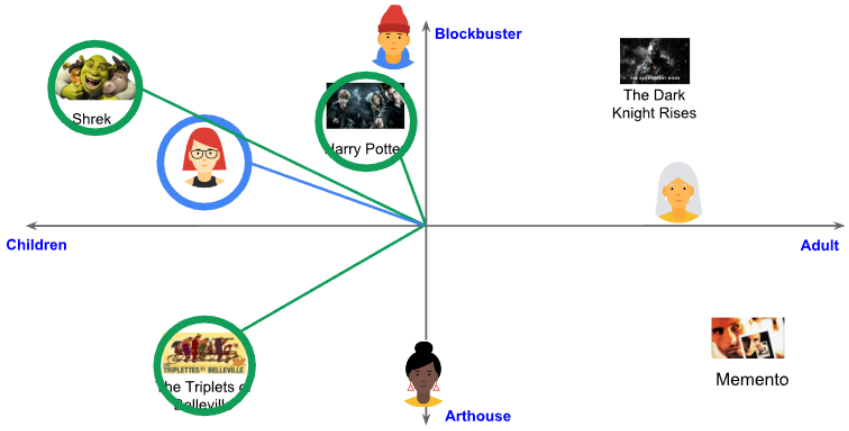


Figura 22: Similaridad en el espacio de embeddings.
Fuente: Google Developers.

2.5.2. Ranking Stage (Scoring)

En esta etapa se toman los resultados del retrieval stage, se evalúan y clasifican los candidatos para seleccionar los ítems que se le recomendaran al usuario. Su tarea es reducir el conjunto de elementos en los que el usuario puede estar interesado a una lista corta de posibles candidatos. Dado que en esta etapa se evalúa un subconjunto relativamente pequeño de elementos, el sistema puede utilizar un modelo más preciso basándose en querys adicionales.

Deep & Cross Network (DCN)
Una DCN es una arquitectura de red neuronal que está diseñada para aprender el cruce de características de los datos [Wang *et al.*, 2017] [Wang *et al.*, 2021], esto es sumamente importante en la etapa de Ranking, por ejemplo si a un usuario le gusta un trago con una bebida Cola y también le gusta otro trago con Ron, el cruce de estas características nos proporciona información adicional sobre las interacciones, por lo que podremos recomendarle un Ron Cola con mayor seguridad, por lo que el cruce de características nos permite generar un modelo más robusto y preciso. La estructura de la DCN es la siguiente:

1. Una capa de input (Generalmente una Capa de Embeddings).
2. Una *Cross Network*, que contiene múltiples capas cruzadas que modelan las interacciones explícitas de las características.
3. Una *Deep Network* que modela las interacciones implícitas de las características.

- **Cross Network:** Este es el núcleo de DCN. Aplica explícitamente el cruce de características en cada capa, y el grado polinomial más alto aumenta con la profundidad de la capa. La siguiente figura muestra la $(i+1)$-ésima cross layer (Capa de cruce).

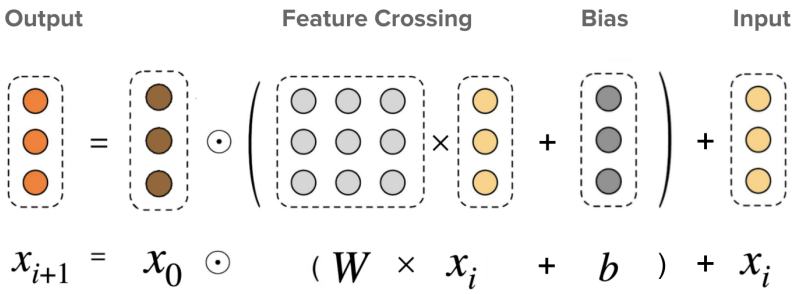


Figura 23: Cross Network.
Fuente: DCN V2, Wang et al.

- **Deep Network:** Es un perceptrón multicapa feedforward (MLP) tradicional.

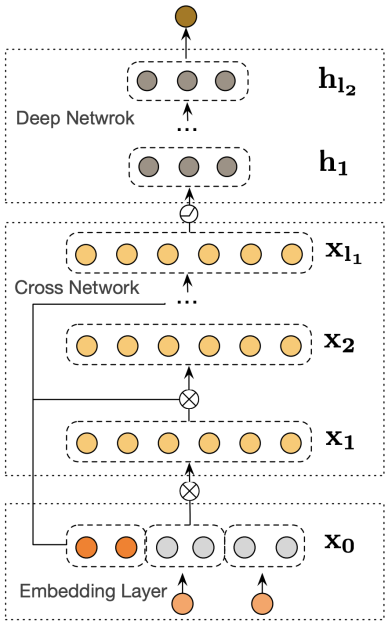


Figura 24: Deep & Cross Network.
Fuente: DCN V2, Wang et al.

2.5.3. Re-Ranking

Finalmente, el sistema debe tener en cuenta restricciones adicionales para la clasificación final, como por ejemplo remover los ítems que el usuario no le gustaron explícitamente. El Re-ranking puede ayudar a asegurar mejor diversidad y equidad en los resultados.

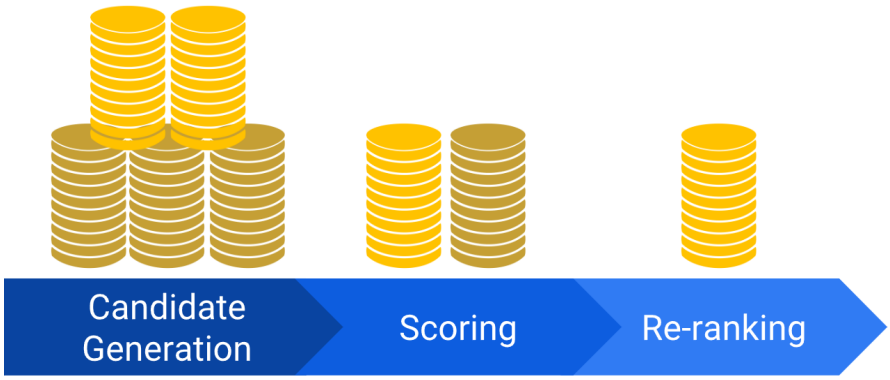


Figura 25: Etapas de un sistema recomendador.
Fuente: Google Developers.

2.5.4. Función de pérdida

Una función de pérdida, o Loss function, es una función que evalúa la desviación entre las predicciones realizadas por la red neuronal y los valores reales de las observaciones utilizadas durante el aprendizaje. Cuanto menor es el resultado de esta función, más eficiente es la red neuronal. Su minimización, es decir, reducir al mínimo la desviación entre el valor predicho y el valor real para una observación dada, se hace ajustando los distintos pesos de la red neuronal. A continuación definiremos algunas funciones de perdidas necesarias para contextualizar el problema.

MSE
Mean Squared Error, como se definió en la sección de métricas (ver 2.1.4) esta métrica también se puede utilizar como función de perdida y es la que utilizaremos para evaluar el rating en la **Deep & Cross Network**, mientras más bajo el MSE mejor serán los resultados.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Donde:

$$n = \text{cantidad de datos},$$
$$Y_i = \text{Valor real},$$
$$\hat{Y}_i = \text{Valor predicho}.$$

Cross Entropy

También llamada logarithmic loss, log loss, o logistic loss, mide la discrepancia entre la distribución de probabilidad predicha por el modelo y la distribución de probabilidad real de las etiquetas o categorías de los datos de entrenamiento. Esta funcion calcula una puntuación o pérdida que penaliza la probabilidad en función de qué tan lejos está del valor esperado real, la penalización es logarítmica y produce una puntuación grande para diferencias grandes cercanas a 1 y una puntuación pequeña para diferencias pequeñas que tienden a 0. Cross-entropy se define como:

$$CE = -\sum_{i=1}^{n} t_i \cdot log(p_i).$$

Donde:

$n =$ cantidad de clases,

$t_i =$ etiqueta real,

$p_i =$ es la probabilidad softmax (ver 2.3.5) para la i-esima clase.

Dependiendo de la cantidad de clases se utiliza o bien **Binary Cross-Entropy** para problemas con dos clases, problemas binarios como puede ser detectar o no una persona, o bien **Categorical Cross-Entropy** cuando se tienen más de dos clases, como por ejemplo detectar si es perro, gato, pájaro u otro animal.

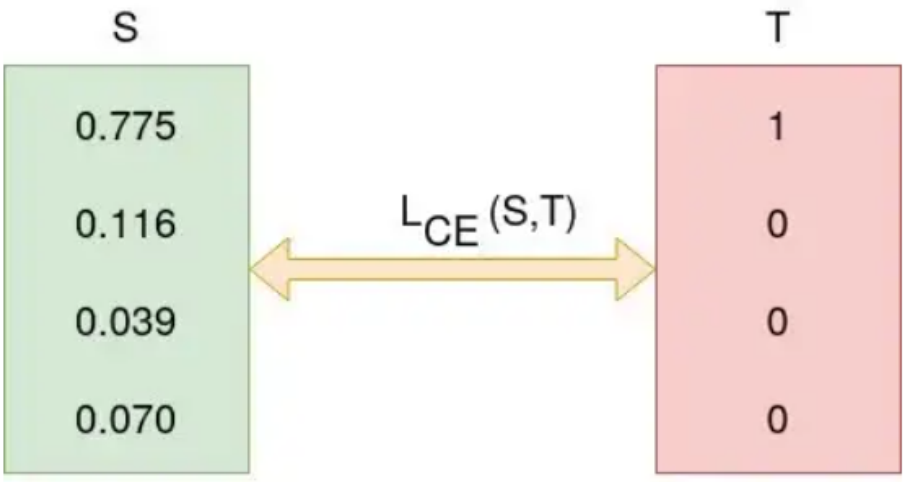


Figura 26: Cross-Entropy de las probabilidades para asignar una clase en específico. Fuente: Kiprono Koech, TowardsDataScience.

Categorical Cross Entropy
se utiliza esta función de perdida cuando tenemos más de 2 clases, la función es la misma que la Cross-Entropy y se utiliza una softmax para asignar una clase u otra, en el caso de los sistemas recomendadores se utiliza esta función para asignar ítem a los correspondientes usuarios que vendrían a actuar como clases, esto ocurre en el Retrieval Stage (Ver sección 2.5.1) entre las queries y los candidatos.

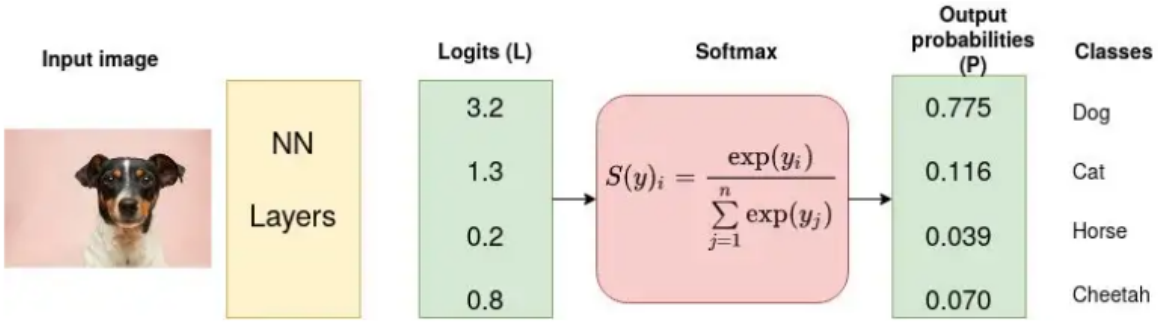


Figura 27: Flujo de probabilidades con Softmax para tarea de clasificación multiclase. Fuente: Kiprono Koech, TowardsDataScience.

TopK
En el caso de los problemas de recomendación no es sencillo definir una función de perdida, ya que la mayoría de veces las interacciones son 1 a 1, y tener una función tan estricta que solo evalué si el modelo recomendó o no determinado ítem es muy complicado, es por esto que se utiliza **TopK Recommendations**, esto se refiere a recomendar K ítems para un usuario dado, si dentro de estos K ítems se encuentra el ítem con el que el usuario interactuó entonces la recomendación es positiva, en el caso contrario la recomendación es negativa, y así obtenemos 2 conjuntos, recomendaciones positivas y negativas, con los cuales podemos calcular accuracy, recall y precisión.

FactorizedTopK
Tensorflow implementa una versión de TopK, llamada FactorizedTopK, esta versión utiliza **TopK Categorical accuracy** con qué frecuencia el ítem con el que el usuario ha interactuado está entre los K principales candidatos predichos para una determinada query. Esta métrica está optimizada para problemas de recomendación y generalmente se utiliza Topk@1, TopK@5, TopK@10, TopK@50 y Topk@100, siendo TopK@1 una penalización muy drástica 1 a 1 y TopK@100 demasiado flexible, por lo que para el sistema recomendador desarrollado se utilizó TopK@10.

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

En este capítulo se describe la propuesta de solución para el problema planteado. Se detalla el análisis de los datos, luego se explican las variables y representaciones a utilizar junto con el preprocesamiento de algunos datos para su correcto uso en el modelo. En la sección de modelamiento, se presentan las diferentes arquitecturas de modelamiento empleadas para resolver el problema de recomendación. Finalmente, en la sección de Herramientas de desarrollo, se detallan los aspectos más técnicos como el lenguaje de programación, librerías, optimizaciones y como se desplegó a producción el sistema recomendador.

3.1. Data

Para esta memoria se trabajó en conjunto con el Club de Campo Naval Las Salinas (CNCS) en Viña del Mar, Chile. Ellos proporcionaron la data de todo un año de los distintos puntos de venta que poseen, tales como la cafetería, el restaurante y el bar, etc. Este dataset contiene **462.110** filas, donde cada fila es la interacción entre un usuario y un producto, tiene **522** productos distintos y **79.474** usuarios distintos. también tiene distintas columnas con información sobre el medio de pago, la hora del pedido, la fecha, etc. Como se puede observar tenemos un dataset bastante complejo, así que se realizó un **EDA** (Exploratory Data Analysis) para ver la distribución de los datos en base a las distintas características.

3.1.1. Distribución de los Datos

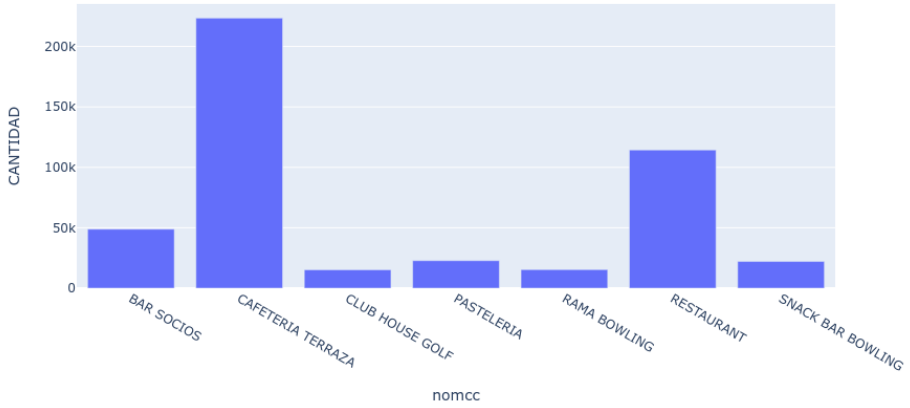


Figura 28: Distribución según puntos de ventas.
Fuente: Propia.

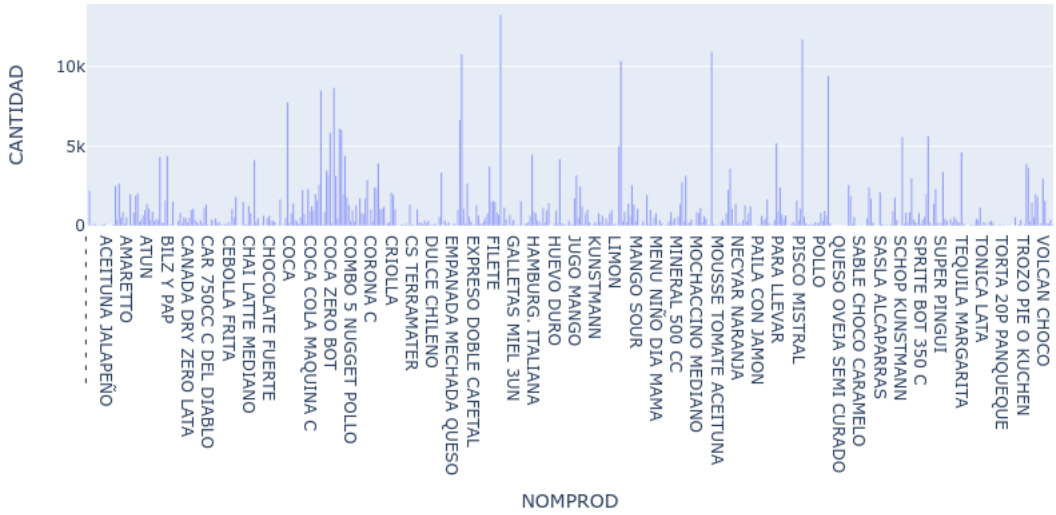


Figura 29: Distribución según productos.
Fuente: Propia.

El Dataset está desbalanceado en la representación de cada producto, ya que hay productos exclusivos de cada punto de venta y hay puntos de venta mucho más concurridos que otros, como lo son **la cafetería** y **el restaurante**.

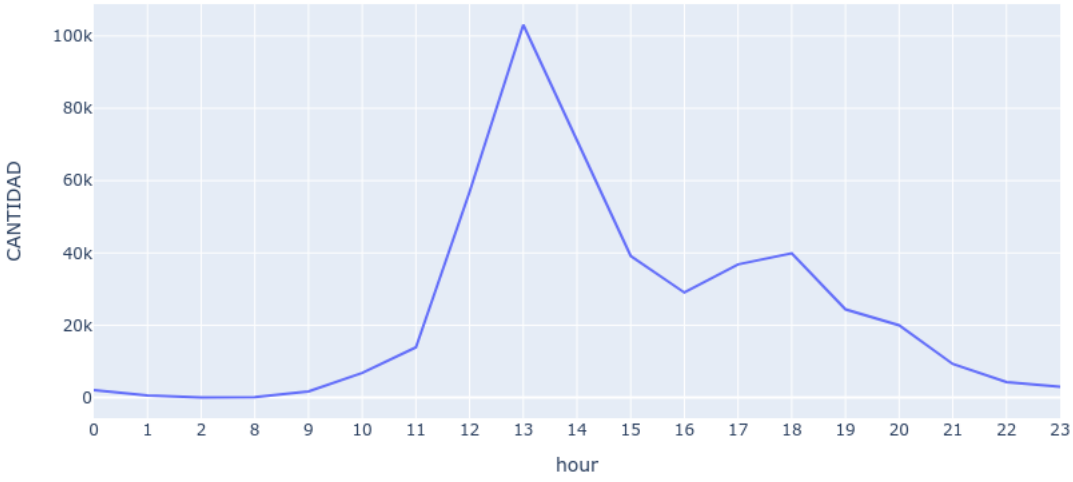


Figura 30: Distribución según hora del pedido.
Fuente: Propia.

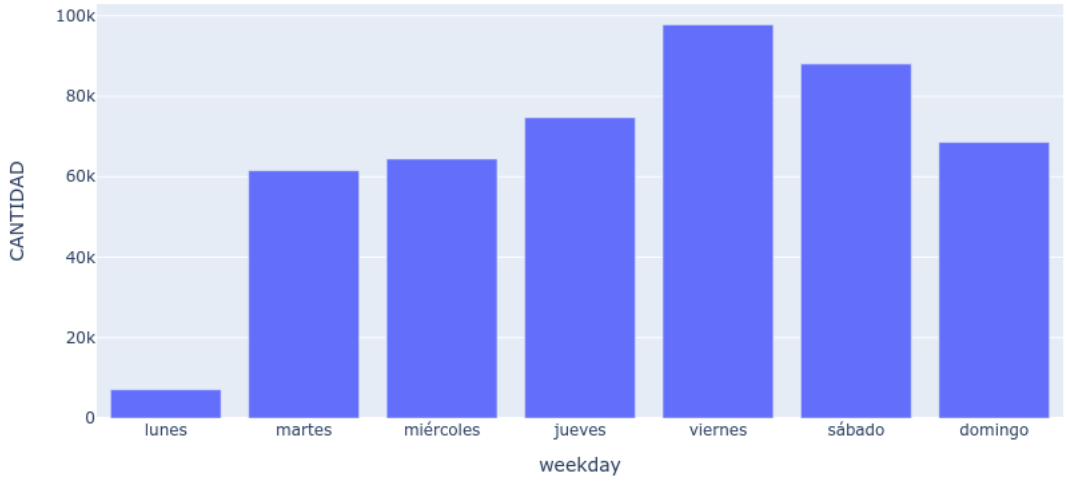


Figura 31: Distribución según día de semana del pedido.
Fuente: Propia.

También se puede analizar que la mayoría de las ventas ocurren a las **13:00** y el día donde más se vende es el **viernes**.

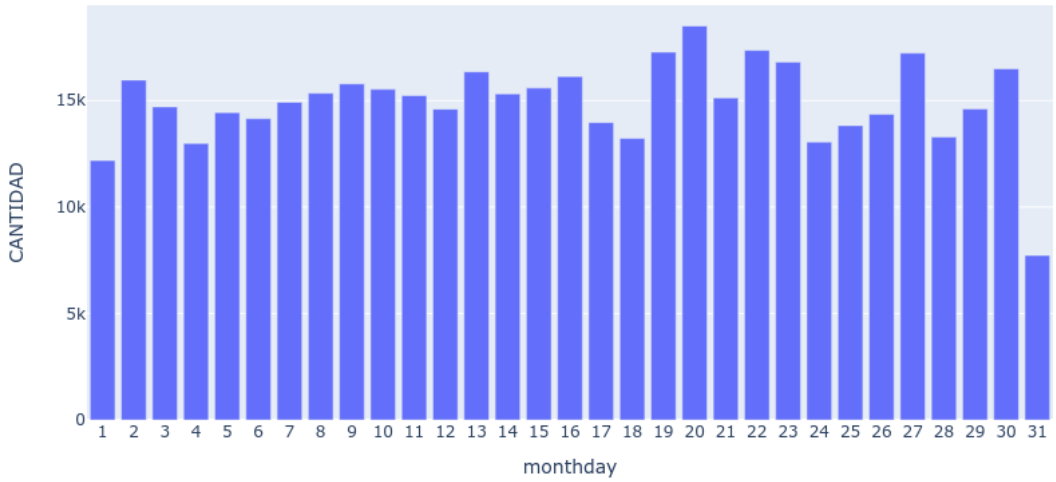


Figura 32: Distribución según día del mes del pedido.
Fuente: Propia.

Si analizamos el día del mes se puede ver como la variación no es tan drástica y podríamos

decir que en promedio se vende lo mismo independiente el número del día del mes.

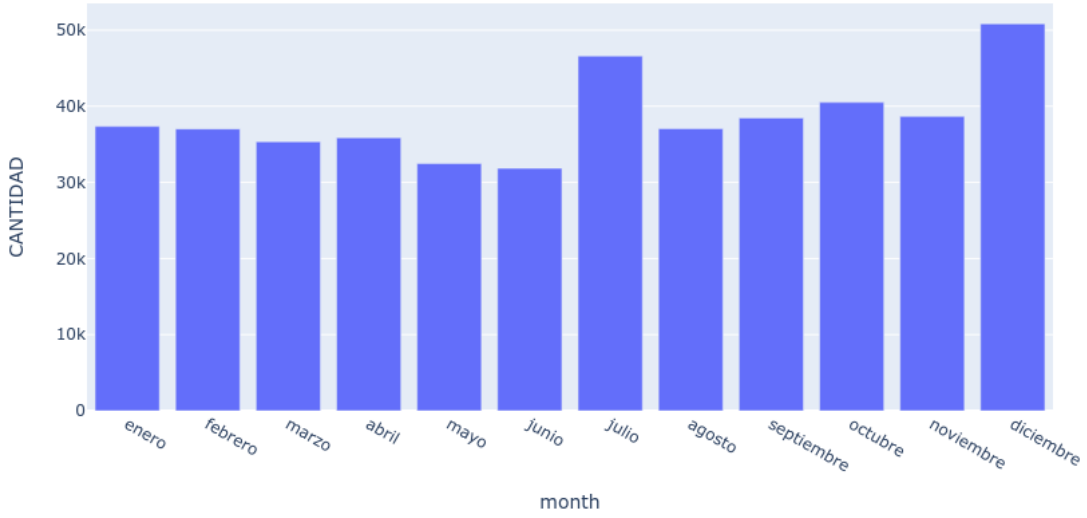


Figura 33: Distribución según mes del pedido.
Fuente: Propia.

Por último es interesante notar la distribución de ventas según los meses, **diciembre** es el mes con más ventas, pero esto podría ser esperable, ya que hay fiestas importantes como navidad y año nuevo dentro de este mes, pero es curioso ver como **julio** es el segundo mes con mayor cantidad de ventas y eso que en Chile no hay una fiesta significativa como podría ser en septiembre la fiesta del 18.

Por temas de confidencialidad de los datos, no se podrá mostrar el análisis en completo, ya que características como el tipo de pago, valores de los productos, etc. podrían comprometer la privacidad de los clientes y del local.

3.1.2. Variables Objetivos

Retrieval:
Para la etapa de Retrieval, la variable objetivo (Ground Truth) es la interacción entre el usuario y el ítem, generaremos una lista de k ítems para un usuario y esperamos que dentro de esos k ítems se encuentre el ítem con el que el usuario realmente interactuó, que se encuentra registrado en el dataset, de ser así la predicción estaría correcta, esto es el **Accuracy TopK** para distintos k.

Ranking:
Por otro lado, para la etapa de Ranking no contamos con la calificación de los usuarios, el restaurante solo guarda el pedido de cada usuario, lo que consumió dentro del local y después tuvo que pagar, esto es un problema, ya que no contamos con los datos necesarios para poder predecir el rating. Para esto se creó una función customizada para generar el rating entre usuarios e ítems.

Para comenzar se creó un diccionario de usuarios y un diccionario de ítems para almacenar las distribuciones correspondientes. En el diccionario de usuarios por cada usuario se guardó en otro diccionario las interacciones que realizo con los ítems, a cada interacción se le dio un peso uniforme, luego ponderamos este peso por la cantidad de veces que el usuario consumió cierto ítem y al final todos los pesos suman 1, es decir, si un usuario a interactuó en total con 4 ítems, los cuales fueron dos mojitos, una hamburguesa y unas papas fritas por ejemplo dentro del dataset, en este diccionario el usuario a tendrá un diccionario propio donde Mojito $= 0{,}50(0{,}25 * 2)$, hamburguesa $= 0{,}25$, papas fritas $= 0{,}25$. La hipótesis es que si a un usuario le gusta un ítem, entonces lo consumirá más veces, ponderando un peso mayor y todos los pesos suman 1 para trabajar con probabilidades.

Después estos a estos pesos se les agregó un ruido generado a través de una distribución normal, donde la media y la desviación estándar provienen del diccionario de ítems, esto agrupado por cada punto de venta del restaurante. En este diccionario por cada ítem se almacena la cantidad de veces que se consumió ese ítem, así podemos saber si un ítem es consumido ampliamente, y utilizaremos este valor normalizado entre $[0, 1]$ dividiendo por el total de interacciones como media de nuestra distribución normal. Para calcular la desviación estándar utilizaremos la cantidad de usuarios **únicos** que consumieron el ítem, también normalizado entre $[0, 1]$ al dividir por el total de usuarios únicos, con esto podemos generar varianza entre estas interacciones.

Este ruido además tiene una ponderación dependiendo de la cantidad de ítems con los que el usuario ha interactuado, a mayor cantidad de ítems, mayor será la ponderación del ruido, la hipótesis detrás de esta ponderación es que si un usuario es muy disperso, no sabrá que es lo que quiere por lo que consumirá ítems distintos para ir probándolos y su varianza será mayor, en cambio, un usuario que ya sabe lo que quiere, consumirá un grupo más reducido de ítems y su varianza será menor. La ponderación del ruido será mucho mayor para un usuario que consume 10 ítems distintos y así generaremos mayor varianza, versus otro usuario que siempre consume los mismos 3 ítems, manteniendo una menor varianza.

$$Ranking(u, i) = UD_u + 0{,}1 \cdot (1 - \frac{1}{len(UD_u)}) \cdot \mathcal{N}(ITD_i,\ (1 - IUD_i)^2) \qquad (1)$$

Donde u es un usuario, i es un ítem, UD es el diccionario de Usuarios que registra las interacciones de cada usuario con sus respectivos ítems, ITD es el diccionario de ítems que registra la interacción de cada ítem con el total de usuarios y IUD el diccionario de ítems que registra la interacción de cada ítem con sus usuarios únicos.
En la sección de **Anexos** (ver página 87), se encuentran los **códigos en Python** para calcular tanto los diccionarios mencionados como la función de Ranking.

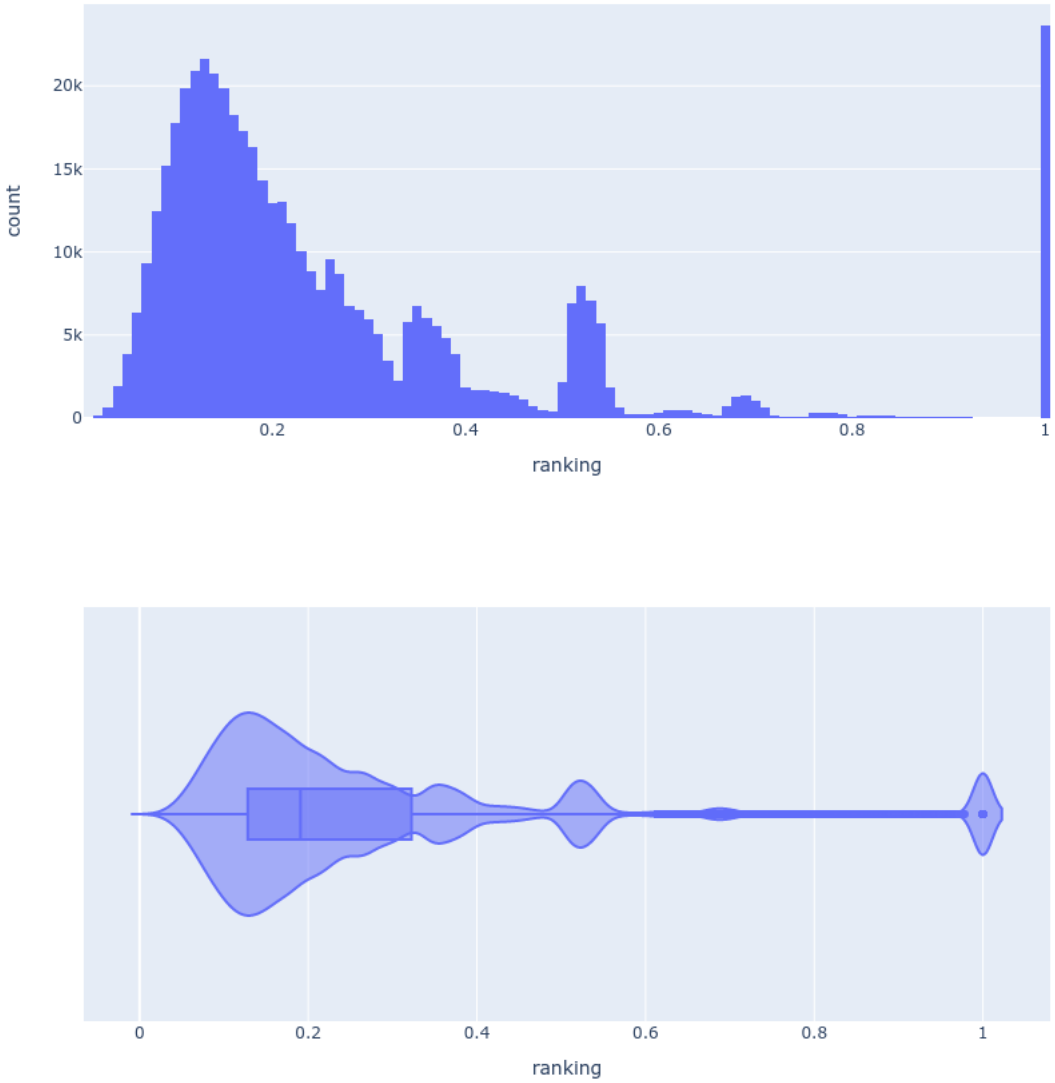


Figura 34: Distribución del Ranking Generado.
Fuente: Propia.

El ranking es un valor continuo entre $[0, 1]$ y las calificaciones de los usuarios en los sistemas de recomendación son discretas, por ejemplo 1,2,3,4,5 estrellas, se aplicó una bucketizacion de los valores del ranking, vamos a trabajar en la aplicación de Cropy con un sistema de estrellas, del 1 al 5, donde cada estrella puede tomar un valor entero o la mitad, por ejemplo 3,5 estrellas, esto nos deja con un intervalo de $[1, 10]$, por lo que la bucketizacion será cada 0.1 en los valores del Ranking, es decir que los valores entre $[0, 0{,}1[$ corresponden a 0.5 estrellas, los valores entre $[0{,}1, 0{,}2[$ corresponden a 1 estrella y así sucesivamente.

3.2. Parámetros, Restricciones y Función Objetivo

Las variables y restricciones propuestas vienen del modelo matemático planteado en la sección 2.1.3 y son las siguientes:

Parámetros

- n: Cantidad de usuarios
- m: cantidad de ítems
- $M_{n,m}$: Matriz de interacciones
- X: Matriz de usuarios, cuyas filas representan los n usuarios
- Y: Matriz de ítems, cuyas filas representan los m ítems.

Restricciones

- Un usuario puede interactuar más de una vez con un ítem, pero se tomará el valor promedio de sus interacciones.

Función Objetivo
Queremos encontrar X e Y tal que minimice el *MSE*

$$(X, Y) = \underset{X,Y}{\text{argmin}} \sum_{(i,j)\in E} [(X_i)(Y_j)^T - M_{ij}]^2. \tag{2}$$

sujeto a factores de regularización implícitos en el modelo como capas de **Dropout** y **Layer Normalization**. Por último, aplicaremos métodos de gradiente descendente y resolveremos el problema mediante un modelo de redes neuronales profundas.

3.3. Representación

Se utilizarán distintas representaciones, cada una dependiendo las etapas del sistema recomendador.

3.3.1. Generación de candidatos

para la etapa de *Retrieval* se utilizarán *Embeddings Multidimensionales* aprendidos por la propia red neuronal, estos embeddings se obtienen a partir de un **Dataframe** en el cual está almacenada toda la información de las interacciones históricas, para la etapa de Retrieval se utilizaran solamente 2 columnas de este Dataframe, las cuales son:

- Username: el nombre del usuario
- Product name: el nombre del ítem,

luego de aislar estos datos, se convertirán a tensores que conformaran el **dataset** de usuarios y productos respectivamente, donde más adelante los modelos consumirán estos datasets en forma de tensores para realizar las operaciones necesarias, el modelo de *Query* se encargara de representar la información de los usuarios y el modelo *Candidate* se encargara de representar los candidatos a recomendar según la información de los ítems.

3.3.2. Ranking

Para la etapa de Ranking, se seguirán utilizando *Embeddings Multidimensionales*, pero ahora con la diferencia que utilizaremos, además del nombre del usuario y del producto, otras características provenientes del **Dataframe** como lo son:

- ranking: la calificación que cada usuario asigna a un ítem, este es el valor más importante y será el que intentaremos predecir en esta etapa.
- Hora del pedido.
- Día de la semana del pedido
- Día del mes del pedido.
- Mes del pedido.

Tanto para la hora, como el día de la semana, el día del mes y los meses, se aplicó una transformación **seno-coseno** para tener representaciones cíclicas, ya que por ejemplo las 01 horas de la mañana está muy lejos numéricamente de las 23 horas, pero nosotros sabemos que están muy cerca a 2 horas de distancia, es por esto que aplicamos la transformación seno-coseno, ya que tanto la hora, como los días de la semana, como los días del mes y los meses del año son cíclicos y queremos preservar esa continuidad entre el inicio y el final de estas secuencias.

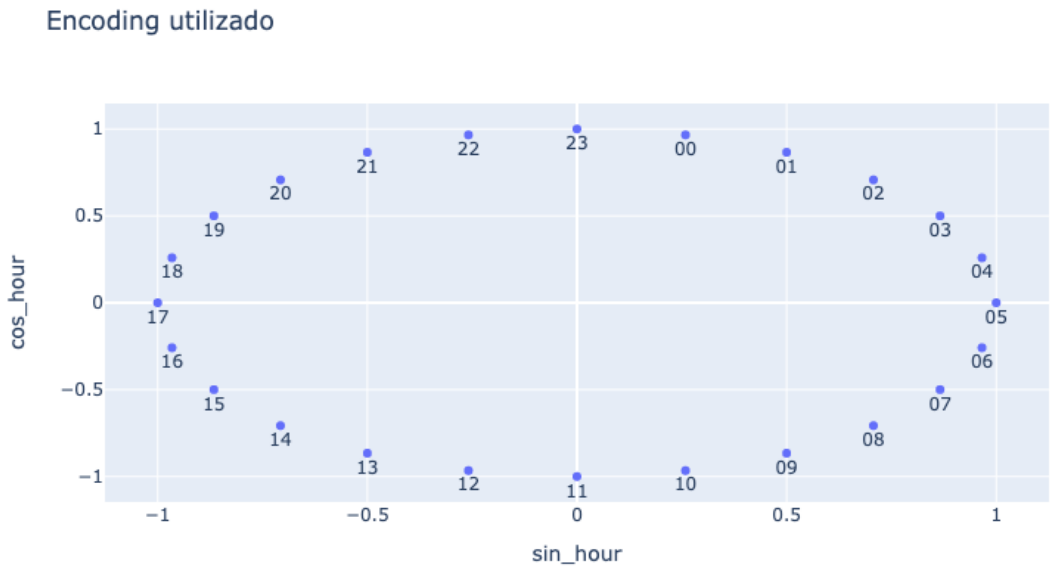


Figura 35: Ejemplo de preprocesamiento para las 24 horas.
Fuente: Propia.

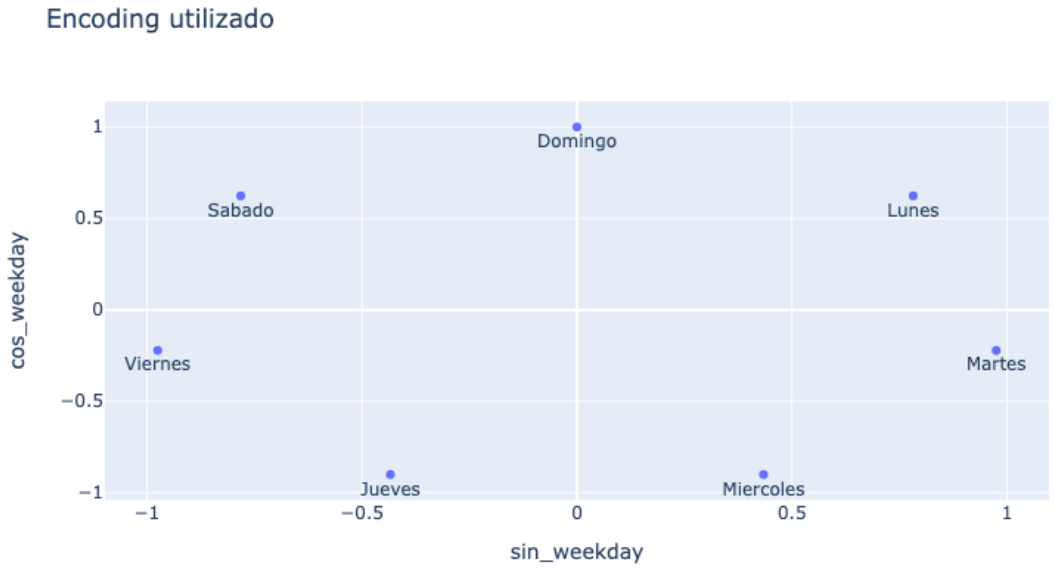


Figura 36: Ejemplo de preprocesamiento los días de la semana.
Fuente: Propia.

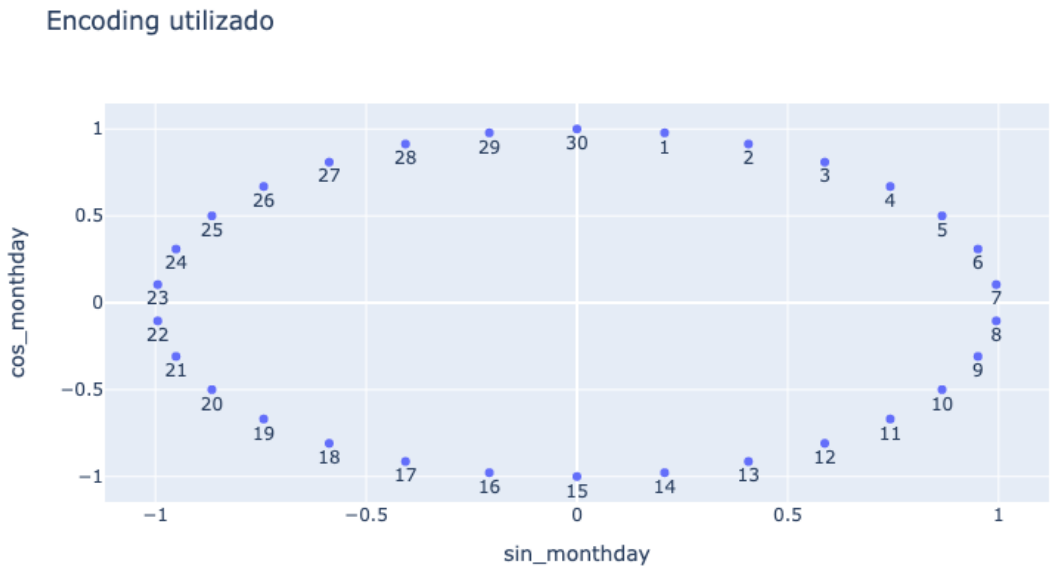


Figura 37: Ejemplo de preprocesamiento para los días del mes.
Fuente: Propia.

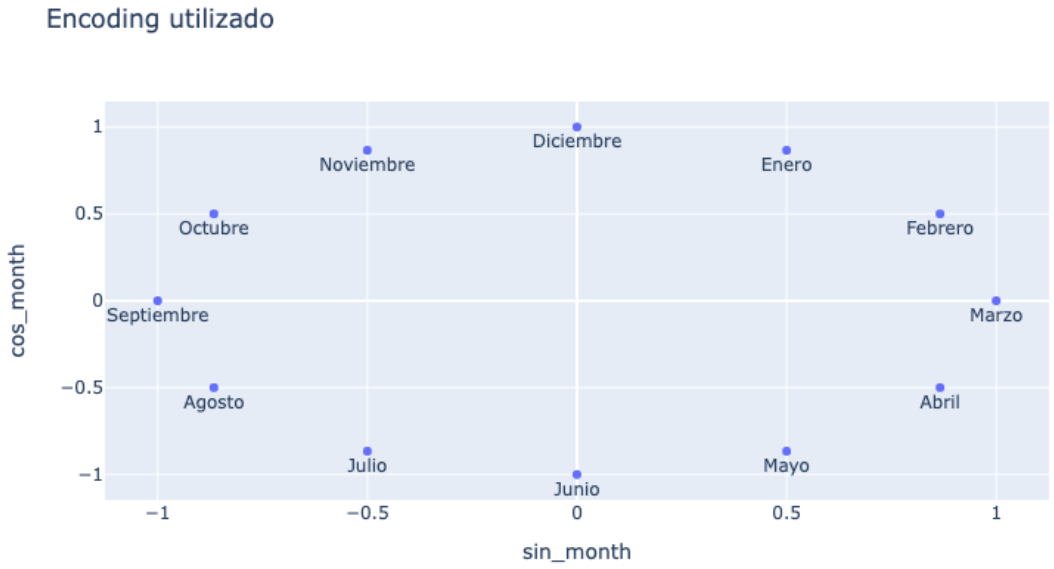


Figura 38: Ejemplo de preprocesamiento para los meses del año.
Fuente: Propia.

Al igual que en la etapa anterior, estos datos se transforman a tensores que conformaran el dataset de Ranking, donde el modelo aprenderá sus representaciones y después se utilizaran en una DCN (Deep & Cross Network) para aprender el cruce de características y obtener una buena predicción del Ranking.

3.4. Modelamiento

Como primera iteración de modelamiento, se plantearan dos modelos básicos, uno para la etapa de retrieval y otro para la etapa de ranking, estos modelos básicos serán los utilizados en la sección de experimentación y dependiendo los resultados se irán modificando, agregando o quitando capas, cambiando funciones de activación, cambiando dimensiones, cantidad de neuronas, etc.

3.4.1. Retrieval Stage

Para el problema de retrieval, se utilizó una arquitectura **Two-Tower**, donde cada torre se encarga de aprender las representaciones y los embeddings de los usuarios y productos respectivamente. Cada Torre recibe los datos en formato de texto y los transforma a un espacio vectorial mediante una **capa de embeddings** entrenable, luego le siguen **bloques de encoder** de Transformers con self-attention y una capa FeedForward. por último agregamos una **capa linear** final para calcular la similitud entre ítems mediante la función **TopK**.

Lo que buscamos con este enfoque es que cada torre pueda aprender y mapear el espacio de embeddings de usuarios y productos respectivamente, para de esta forma aprender las similitudes entre distintos productos y distintos usuarios, así la información de productos similares (Filtrado basado en Contenido, sección 2.1.1) o usuarios similares (Filtrado Colaborativo, sección 2.1.1) pueden ayudar a la red a generar mejores recomendaciones, es una forma de mezclar los dos enfoques de recomendaciones mediante el uso de Transformers y métodos de atención.

3.4.2. Ranking Stage

Para el problema de Ranking, tenemos al igual que antes una capa de embeddings entrenable y una **Cross-Net** donde se calcula el cruce de todas las características de los ítems, para luego pasar por una serie de **capas densas** y finalmente una **capa linear** para calcular el **ranking**, se utilizó la función de perdida **MSE** (Mean Squared Error) para comparar las predicciones con el ranking real.

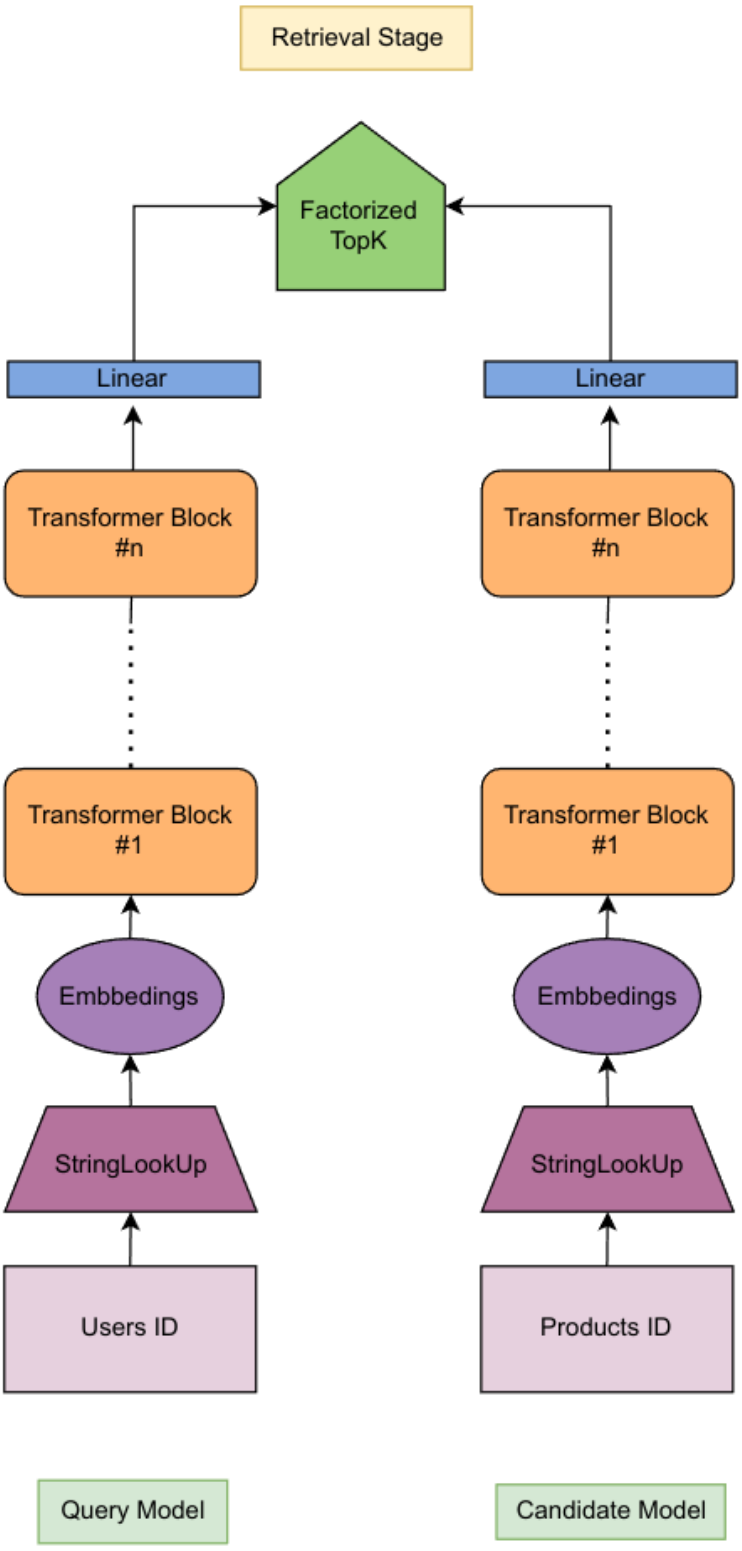


Figura 39: Retrieval Stage.
Fuente: Propia.

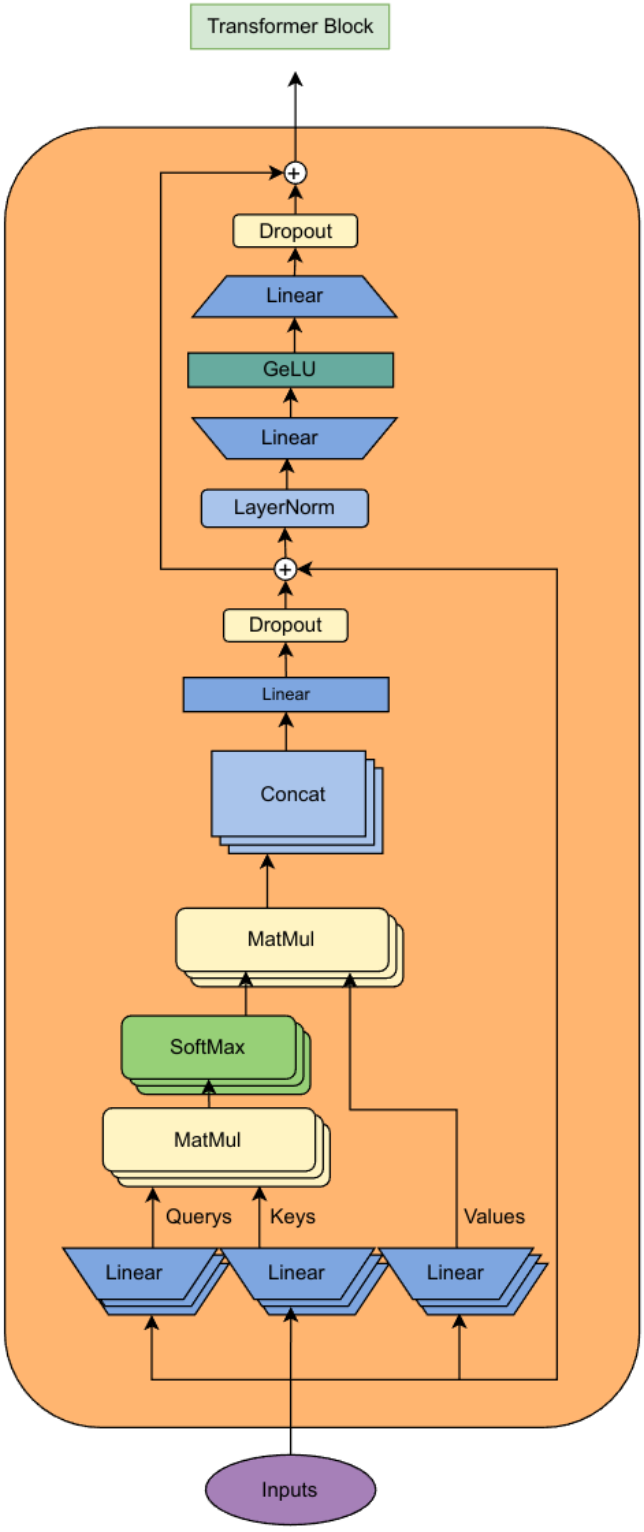


Figura 40: Bloque Transformer.
Fuente: Propia.

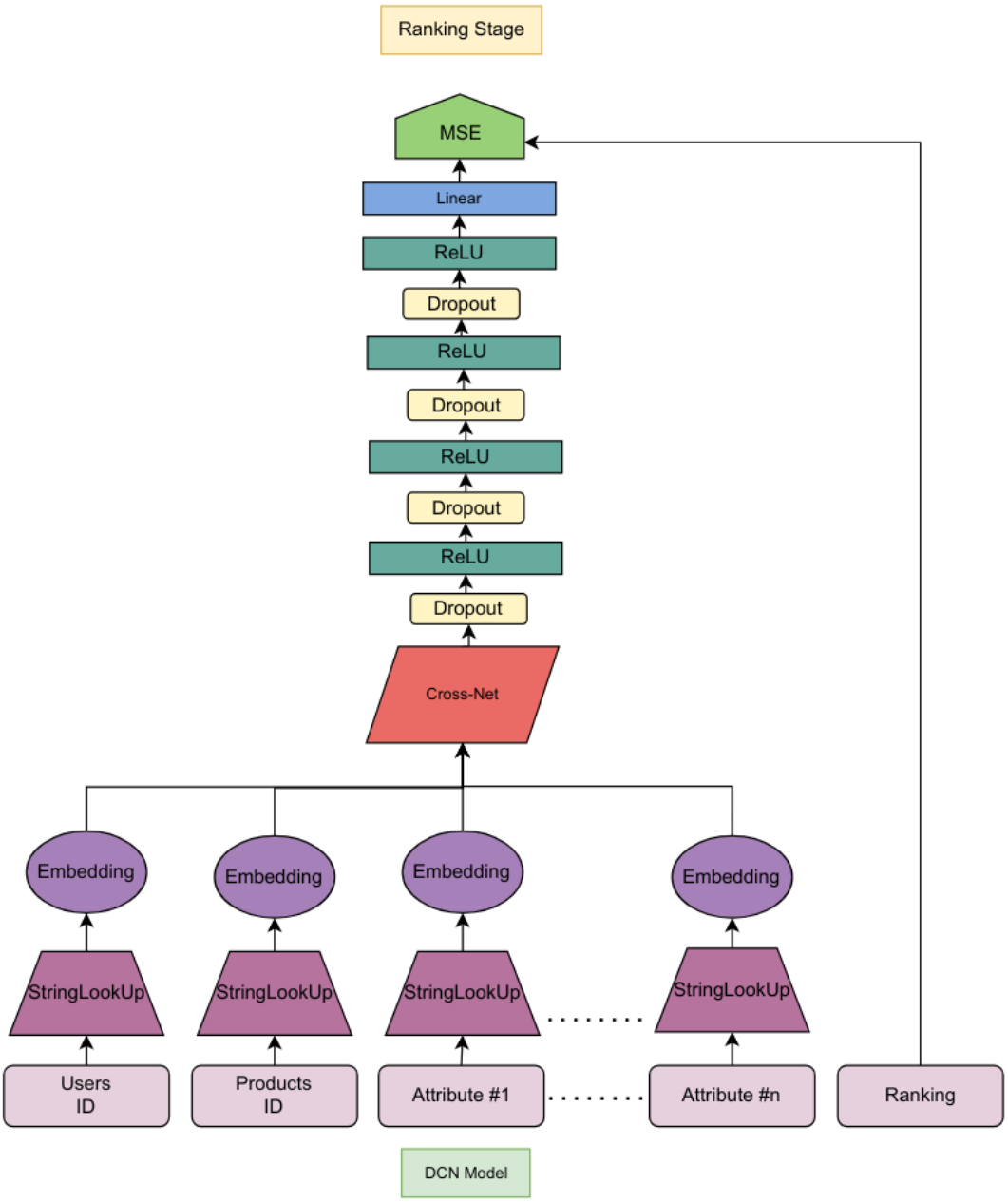


Figura 41: Ranking Stage.
Fuente: Propia.

3.5. Herramientas de Desarrollo

3.5.1. Lenguaje de Programación

El lenguaje de programación que se escogió fue **Python**, es un lenguaje de programación multipropósito y es el lenguaje más utilizado para realizar Machine Learning y desarrollar modelos de redes neuronales, las dos librerías más famosas para esto son **PyTorch** y **TensorFlow** (Ver [StackOverflow, 2022] y [Kaggle, 2022]). Se utilizará Python tanto como para leer los datos, realizar el preprocesamiento y también para crear y entrenar los modelos recomendadores.

3.5.2. Librerias

A continuación se dará un breve repaso sobre las librerías utilizadas en la memoria y su utilidad.

- **NumPy:** es una librería para Python que se enfoca en la computación científica, agrega soporte para trabajar con arreglos multidimensionales (vectores, matrices, Etc.) y además provee operadores optimizados para realizar cálculo y álgebra de manera muy rápida en Python.
- **Pandas:** es una librería para Python que se centra en el análisis y manipulación de datos, permite poder leer y trabajar con archivos de data tabular como CSV, Excel, etc.
- **Plotly y Matplotlib:** Tanto Plotly como Matplotlib son librerías para Python desarrolladas para crear visualizaciones gráficas de los datos, como lo pueden ser gráficos de barras, histogramas, matrices de confusión, etc. Una de las grandes diferencias entre Matplotlib y Plotly es que este último permite crear gráficos interactivos entregando mayor información.
- **TensorFlow:** es una librería para Python desarrollada por Google, enfocada a la creación de modelos de machine learning y deep learning. Si bien es una librería para Python, TensorFlow está desarrollado en su mayoría en C++ y CUDA, ya que esta librería tiene acceso a aceleración por hardware en GPU o TPU.
- **Keras:** es una API que viene integrada con tensorflow y está diseñada para ser muy fácil de entender y de programar, está diseñada para seres humanos, no para máquinas. Keras sigue las mejores prácticas para reducir la carga cognitiva y ser muy simple y amigable para el programado
- **TensorFlow Recommenders (TFRS):** es una librería que se extiende de tensorflow, y está desarrollada con el objetivo de crear modelos de sistemas de recomendación.

Ayuda con el flujo de trabajo completo de la construcción de un sistema de recomendación: preparación de datos, formulación de modelos, entrenamiento, evaluación e implementación.

- **ScaNN:** es una librería de Google Research que realiza búsquedas de similitud de vectores densos a gran escala. Dada una base de datos de embeddings candidatos, ScaNN indexa estos embeddings de una manera que permite buscarlos rápidamente en el momento de la inferencia. ScanN utiliza técnicas de compresión de vectores y algoritmos optimizados para lograr el mejor equilibrio entre velocidad y precisión. Puede superar en gran medida la búsqueda de fuerza bruta y sacrificar poco en términos de precisión.

3.5.3. Despliegue del modelo

Para hacer el despliegue de los modelos del sistema de recomendación se utilizó **Docker**. Docker es una plataforma de código abierto que permite crear y ejecutar aplicaciones en contenedores, los cuales son entornos aislados y portátiles que contienen todo lo necesario para que la aplicación funcione de manera consistente en diferentes sistemas operativos y entornos de ejecución.

Para el modelo de Retrieval se utilizó una imagen de **TensorFlow-Scann** diseñada y optimizada para desplegar modelos de recomendación, creada por Google

Mediante estos contenedores se pueden realizar peticiones POST HTTP a través del formato JSON para obtener las distintas recomendaciones según el ID de un usuario.

3.5.4. Optimizaciones

Los modelos de redes neuronales son bastante más complejos que los modelos tradicionales de machine learning, es por esto que al trabajar con grandes cantidades de datos es necesario optimizar las pipelines, los modelos y el formato en el que provienen los datos, para esta memoria se aplicaron optimizaciones a distintos niveles de desarrollo.

Optimización de Datos

- **Parquet:** es un formato de archivo de almacenamiento en columnas, lo que permite una mejor compresión y una lectura más eficiente de los datos. para optimizar los tiempos de lectura y el espacio de los datos, todos los datasets se convirtieron de formato **.CSV** a formato **.Parquet**.

- **Arrow y Pyarrow:** Arrow es una especificación de formato de datos de código abierto que proporciona un conjunto de estructuras de datos y un esquema para representar datos de manera eficiente en memoria y en disco. Se utilizó el backend de Arrow en Python y pandas con **Pyarrow** para aumentar la velocidad de las funciones de lectura de datos.

Optimización de Pipeline

- **Cache:** es una función que permite almacenar en caché los datos en la memoria, lo que permite una carga más rápida y eficiente de los datos en iteraciones posteriores y es especialmente útil cuando se trabaja con conjuntos de datos grandes que no caben completamente en la memoria.
- **Prefetch:** es una función que permite la carga asíncrona de datos en memoria caché mientras el modelo de aprendizaje profundo procesa los datos actuales.
- **Parallel Calls:** es una función que permite cargar y pre procesar datos en paralelo utilizando múltiples núcleos de CPU, lo que permite una carga y procesamiento más eficiente de los datos.
- **AUTOTUNE:** es un método para ajustar dinámicamente el número de llamadas en paralelo que se realizan durante la carga y preprocesamiento de datos en función del hardware y de la carga de trabajo específicos

3.6. Repositorio

Todo el código y la implementación de los distintos modelos está disponible en un repositorio propio con licencia **GNU GPLv3**, para que otras personas puedan leer el código, utilizar la arquitectura desarrollada en esta memoria y aportar mejoras, con esto se busca aportar a la comunidad Open Source. Así mismo, el repositorio de GitHub servirá para poder replicar los experimentos y reproducir los resultados.

Estos dos puntos son muy importantes en el trabajo de la memoria, ya que, por un lado, aportar a la comunidad open source y liberar el conocimiento aplicado es una gran forma de impulsar la investigación y desarrollo de nuevas tecnologías y productos, por otro parte, el proveer las herramientas necesarias para poder reproducir y replicar los experimentos y resultados también es una parte importante para el ámbito de la investigación, para comprobar los resultados y sentar un baseline para futuras investigaciones.

- **Repositorio Github:** Cropy-Recommender-System

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

4.1. Metodologia

Para que el sistema recomendador sea una solución óptima al problema planteado, tiene que ser validado según una serie de experimentos propuestos más adelante, esto significa que el modelo sufrirá modificaciones con tal de adecuarse al contexto necesario, para esto la metodología utilizada será la de plantear un modelo básico que servirá como esqueleto para el modelo final, este modelo básico se evaluara frente a una serie de experimentos bajo el mismo dataset, dividido en train, validation y test manteniendo la misma semilla para preservar la reproducibilidad de los experimentos y comparar los modelos en condiciones justas. Se evaluará el desempeño del validation set, al terminar la experimentación utilizaremos el modelo con mejor desempeño y será con el que entrenemos la versión final del sistema recomendador. Los siguientes experimentos y optimización de hiperparámetros se realizaron tomando en cuenta los papers *Training Tips for the Transformer Model* [Popel y Bojar, 2018] y *Efficient Transformers: A Survey* [Tay *et al.*, 2022].

Se evaluarán distintos hiperparámetros tales como:

- Optimizador
- Learning Rate
- Arquitectura del Transformer

4.2. Experimentos

4.2.1. Entorno de Experimentacion

El entrenamiento de los modelos se realizó en el siguiente entorno local:

- **Procesador:** AMD Ryzen 7 5800H con 8 Cores y 16 Threads a 3.2 Ghz.
- **RAM:** 16 GB DDR4 a 3200 Mhz.
- **GPU:** Nvidia RTX 3070 Mobile con 8Gb VRAM y Compute Capability 8.6.
- **OS:** Linux Pop!_OS 21.04.
- **Software:** Python 3.11, Nvidia Driver Version 535.86.05, CUDA 12.2, CUDNN 8.6, TensorFlow 2.11.

Para todos los entrenamientos, se utilizó la misma semilla (42) y se separó el dataset en 80 % para entrenamiento, 10 % para validación, y 10 % para testing, estratificado por las distintas clases, en este caso las clases son los productos, para que así cada conjunto de datos tenga una distribución equitativa de ejemplos.

También se utilizarán callbacks para monitorear el desempeño de los modelos junto con **Early Stopping**, para detener el entrenamiento después de una serie de epochs definidas por el parámetro de paciencia, en el cual la función de perdida no haya disminuido.

Por último, la cantidad de neuronas y en general la cantidad de parámetros entrenables va a estar sujeta a una cota superior, la cual no va a poder ser superada, esta decisión está basada en una serie de *Rules of Thumb* (Reglas de oro) en machine learning, hay que aclarar que estas reglas se siguen como buenas costumbres, pero no tienen un fundamento estadístico o matemático profundo detrás de ellas, simplemente son recomendaciones que ayudan a mejorar el desempeño de los modelos de machine learning.

Estas reglas que se incorporaran tienen en cuenta la cantidad de parámetros de la red y la dimensión de los embeddings

- **Cantidad de Parámetros Entrenables:** para determinar la cota máxima de parámetros entrenables se utilizó la siguiente fórmula:

$$d \leq \frac{M}{10}. \tag{3}$$

  donde d es la cantidad de parámetros entrenables y M es el tamaño de nuestro dataset. Esta decisión se tomó para reducir el overfitting, es común notar que se recomienda que la cantidad de data para entrenar a modelos de machine learning sea de al menos un orden de magnitud superior al de parámetros del modelo.

- **Dimensión de Embedding:** para determinar la cota máxima de la dimensión de los embeddings se utilizó la siguiente fórmula:

$$D_E = |C_M|^{\frac{1}{4}}. \tag{4}$$

  donde D_E es la dimensión del embedding, y $|C_M|$ es la cardinalidad de las clases del dataset, esto quiere decir que la cota máxima para los embeddings debe ser la raíz cuarta del número de categorías, esta regla se obtuvo del blog de Google (Google Blog for Developers)

4.3. Resultados

4.3.1. Retrieval Stage

Primero evaluaremos la etapa de Retrieval, como métrica principal se utilizara **categorical accuracy topK@10** y también como función de perdida **Categorical Cross-Entropy**.

Si bien en muchos sistemas de recomendación la métrica más utilizada es **Mean Average Precision** (mAP), en este caso el dataset provisto por el Club Naval de Campo Las Salinas viene estructurado de tal manera que cada producto está asignado a un usuario, esta asignación 1 a 1 significa que hay solo ground truth para cada query, por lo que en este caso al tener un accuracy binario (esta o no está presente el ground truth ítem en la query), el topK Accuracy es lo mismo que **Recall@K**.

En este caso se calcularon distintos TopK Accuracy o lo que es lo mismo Recall@K, con distintos K, esto quiere decir que verificaremos si para una query de largo K, está o no está presente el producto que el usuario consumió, con $K = \{1, 5, 10, 50, 100\}$, hay que dejar en claro que mientras más alto sea el K, más probabilidades hay de que el ítem correcto se encuentre en la query, pero no nos interesa devolver una query con 100 productos, ya que es bastante ineficiente y prácticamente estaríamos recomendando todos los productos, esta no es una solución aceptable, por lo que nos enfocaremos en optimizar la métrica accuracy topK@10 o Recall@10, es decir si está presente el ítem que el usuario consumió dentro de una query de largo 10, este tamaño es aceptable y tampoco penaliza tan drásticamente al sistema recomendador como lo haría una query de largo 5 o largo 1.

Ejemplo de Recall y Accuracy:

Supongamos que tenemos estamos evaluando Accuracy/Recall, eso significa que cada predicción retornará una lista de largo K con distintos productos, para un usuario el ground truth o el valor verdadero será el ítem C, entonces el modelo de retrieval de nuestro sistema recomendador retorna la siguiente predicción para distintos K

- Con K = 1 $\rightarrow y_{pred} = [A]$.
- Con K = 5 $\rightarrow y_{pred} = [A, F, K, L, O]$.
- Con K = 10 $\rightarrow y_{pred} = [A, F, K, L, O, T, C, R, Z, N]$.

y así sucesivamente para K $>$ 10, ahora procederemos a calcular el acuraccy/recall para cada K

- $y_{true} = [C]$
- Con K = 1 $\rightarrow y_{pred}$ = [0] $\rightarrow$Recall@1 = 0.
- Con K = 1 $\rightarrow y_{pred}$ = [0,0,0,0,0] $\rightarrow$Recall@5 = 0.
- Con K = 1 $\rightarrow y_{pred}$ = [0,0,0,0,0,0,1,0,0,0] $\rightarrow$Recall@10 = 1.

Utilizamos Recall y no Precisión, ya que no nos interesa aún el orden de los productos, porque los productos se van a ordenar en la etapa de Ranking del sistema recomendador, en esta etapa lo único que nos interesa es poder recuperar eficientemente los productos con los que el usuario podría estar interesado y así en la etapa siguiente poder ordenarlos por score y prioridad para el usuario.

Modelo Base

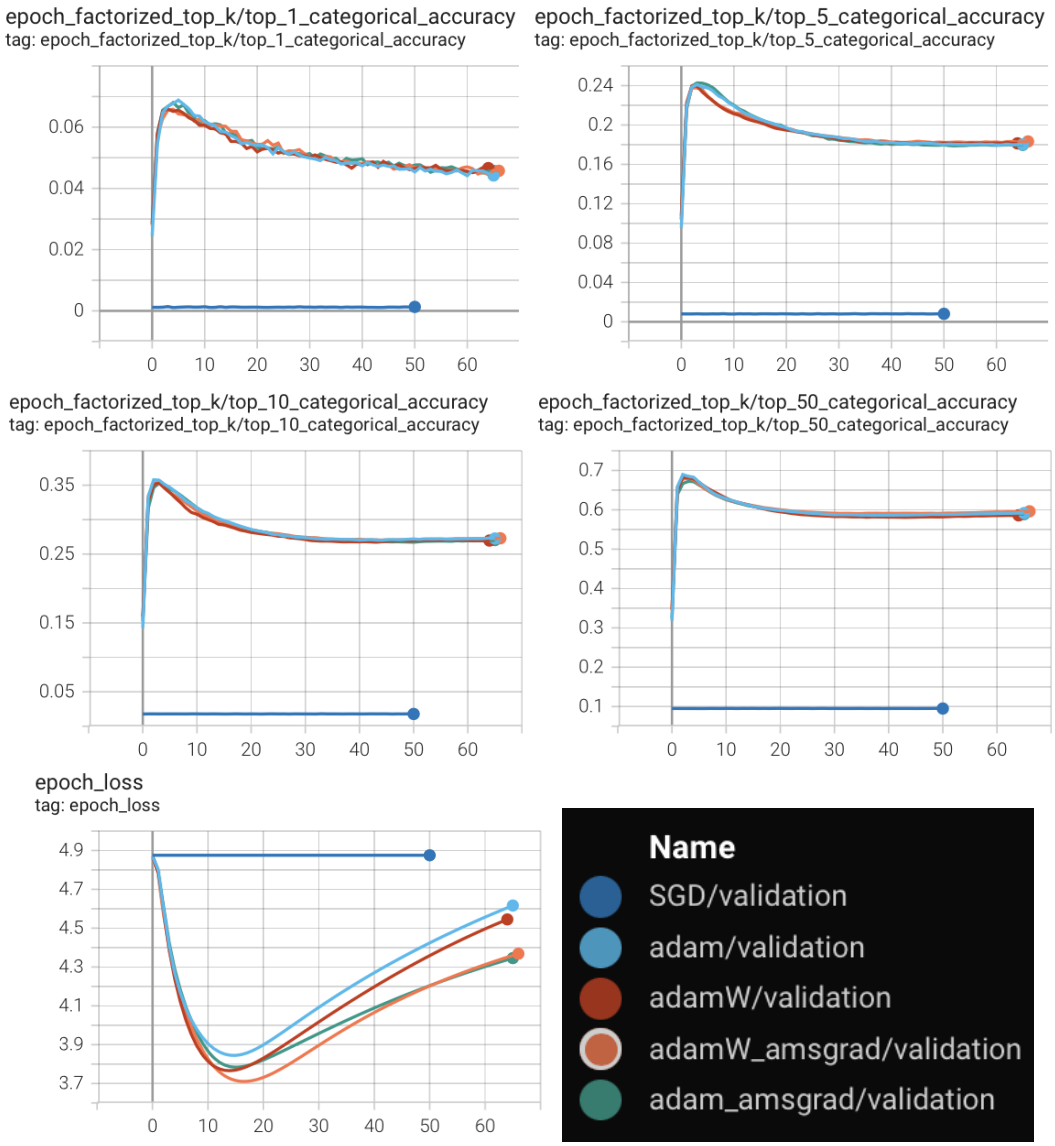


Figura 42: Accuracy TopK@10 y Loss para distintos Optimizadores con la arquitectura base. Fuente: Elaboración Propia.

Optimizer	Recall@10	Loss
SGD	0.0178	4.875
Adam	0.3574	3.845
Adam+Amsgrad	0.3525	3.784
AdamW	0.3535	3.769
AdamW+Amsgrad	**0.3529**	**3.715**

Tabla 2: Tabla de Resultados Recall@10 y Categorical Cross Entropy

Deep Model

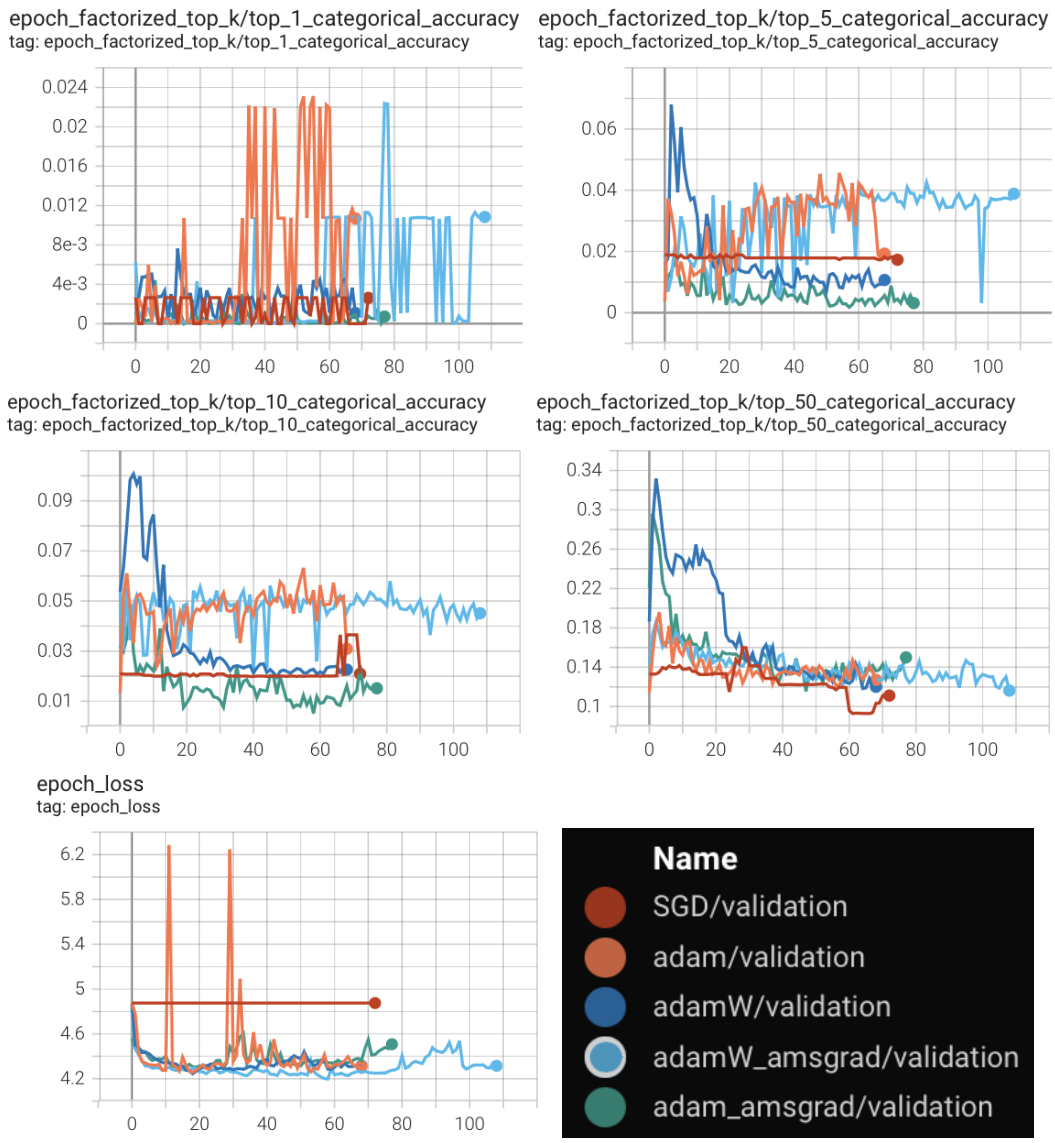


Figura 43: Accuracy TopK@10 y Loss con la arquitectura Base con más capas profundas. Fuente: Elaboración Propia.

Optimizer	Recall@10	Loss
SGD	0.0364	4.875
Adam	0.0632	4.274
Adam+Amsgrad	0.0489	4.293
AdamW	0.1007	4.244
AdamW+Amsgrad	**0.0578**	**4.197**

Tabla 3: Tabla de Resultados Recall@10 y Categorical Cross Entropy

Pre-LN Transformer

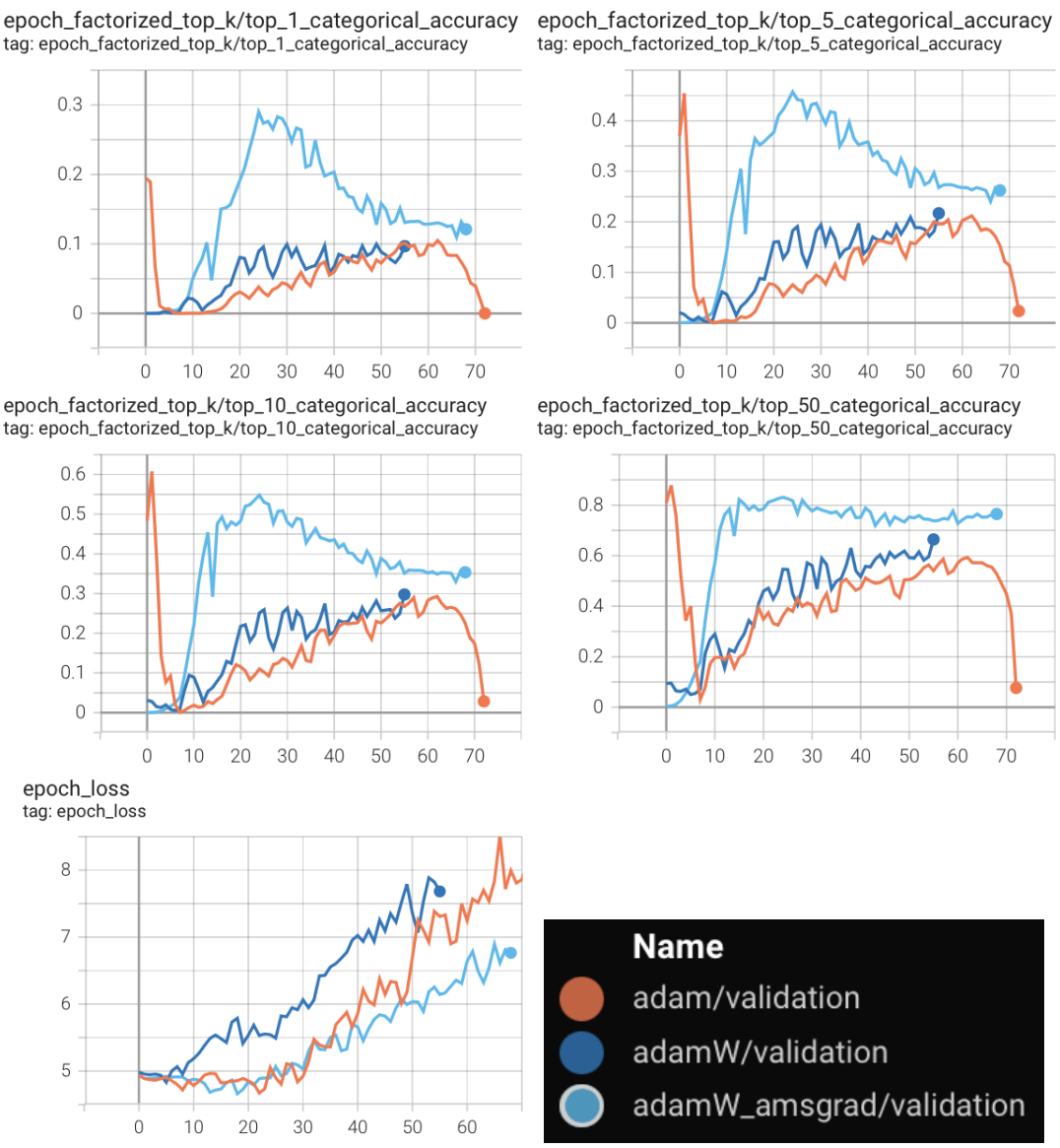


Figura 44: Accuracy TopK@10 y Loss para distintos Optimizadores con la arquitectura Pre-LN.

Fuente: Elaboración Propia.

Optimizer	Recall@10	Loss
Adam	0.2929	4.671
AdamW	0.2976	4.83
AdamW+Amsgrad	**0.5479**	**4.662**

Tabla 4: Tabla de Resultados Recall@10 y Categorical Cross Entropy

Post-LN Transformer

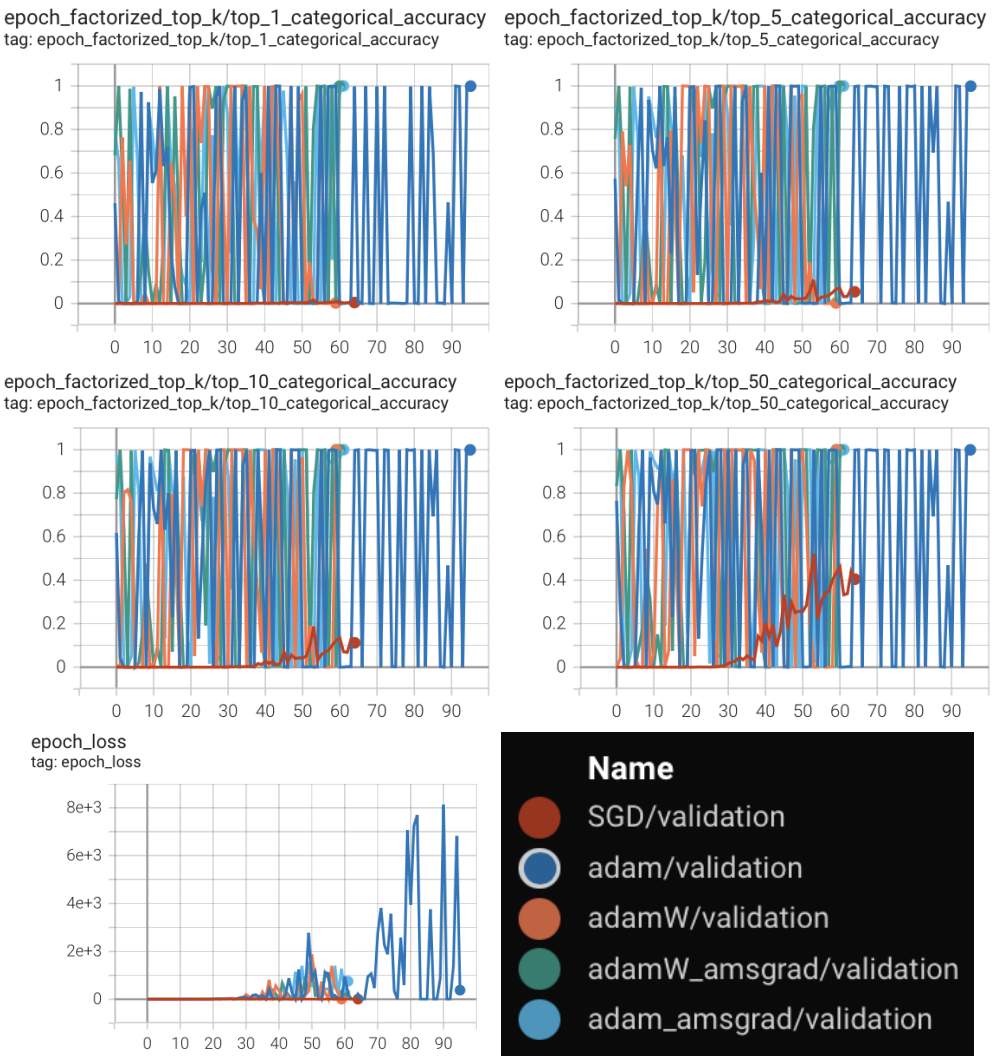


Figura 45: Accuracy TopK@10 y Loss para distintos Optimizadores con la arquitectura Post-LN.

Fuente: Elaboración Propia.

Como se puede observar en las curvas de entrenamiento, el modelo con la arquitectura Transformer de Post layer Normalization es muy errático, cabe destacar que se trabajó con el scheduled learning rate original del paper *Attention is all you need* [Vaswani *et al.*, 2017] y aun así no se logró un entrenamiento adecuado.

Tanto en el modelo base, como en el más profundo y en el modelo con Transformers con Pre-Layer Normalization, **Adam+Amsgrad** es el optimizador que mejor funciona, por lo que para el siguiente experimento se trabajará solo con ese optimizador y distintos learning rates.

Learning Rate

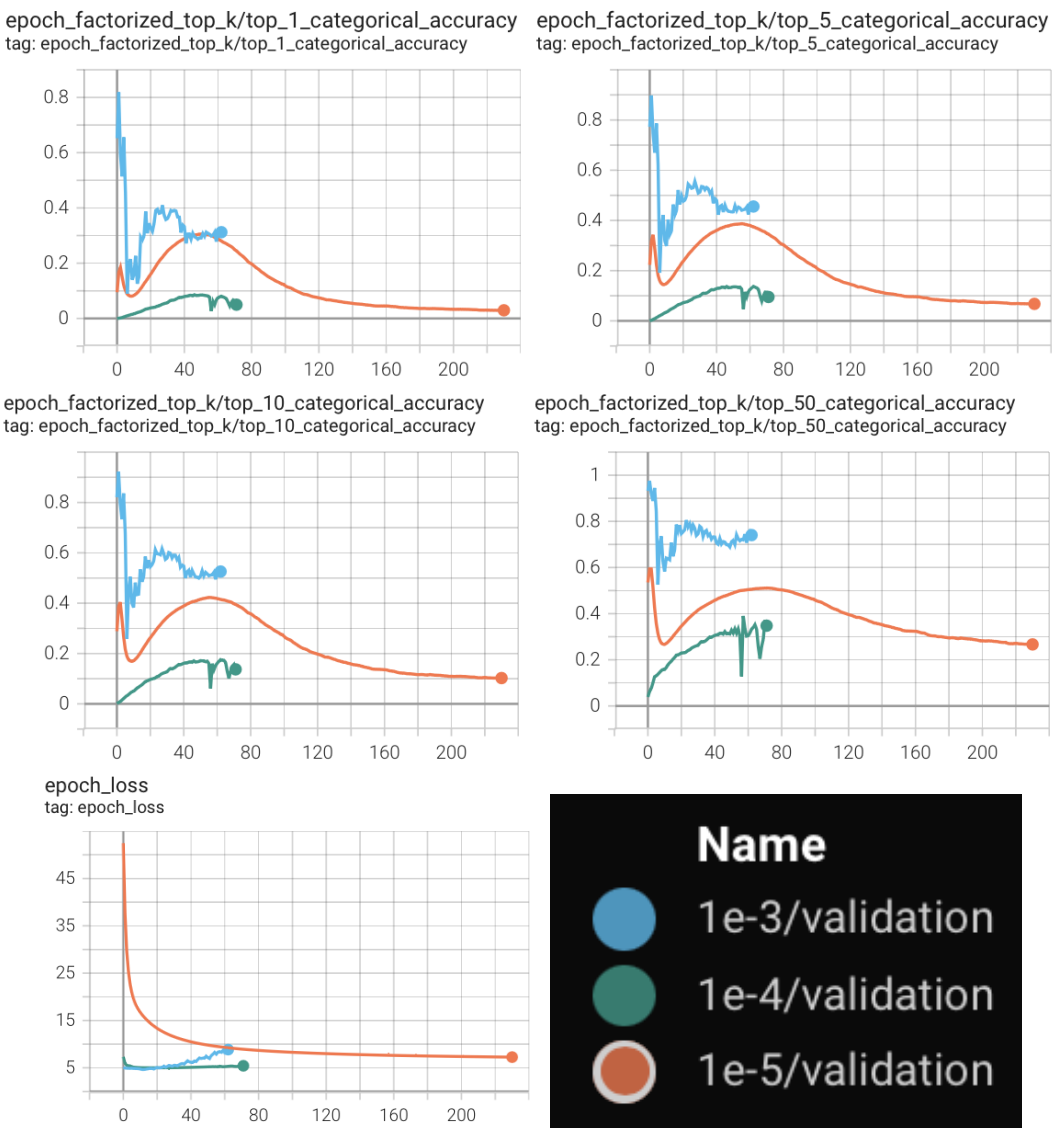


Figura 46: Accuracy TopK@10 y Loss para distintos Learning Rates.
Fuente: Elaboración Propia.

Learning Rate	Recall@10	Loss
1e-3	**0.6159**	**4.727**
1e-4	0.1755	5.001
1e-5	0.4229	7.253

Tabla 5: Tabla de Resultados Recall@10 y Categorical Cross Entropy

Attention Stack

Para el siguiente experimento se dejó fijo el learning rate en **1e-3** que dio mejores resultados. Se decidió aplicar la cota máxima de parámetros entrenables para cada torre por separado y ambas torres a la vez, por eso se disminuyo la cantidad de stacks de encoders a la mitad y la disminución de parámetros entregó mejores resultados.

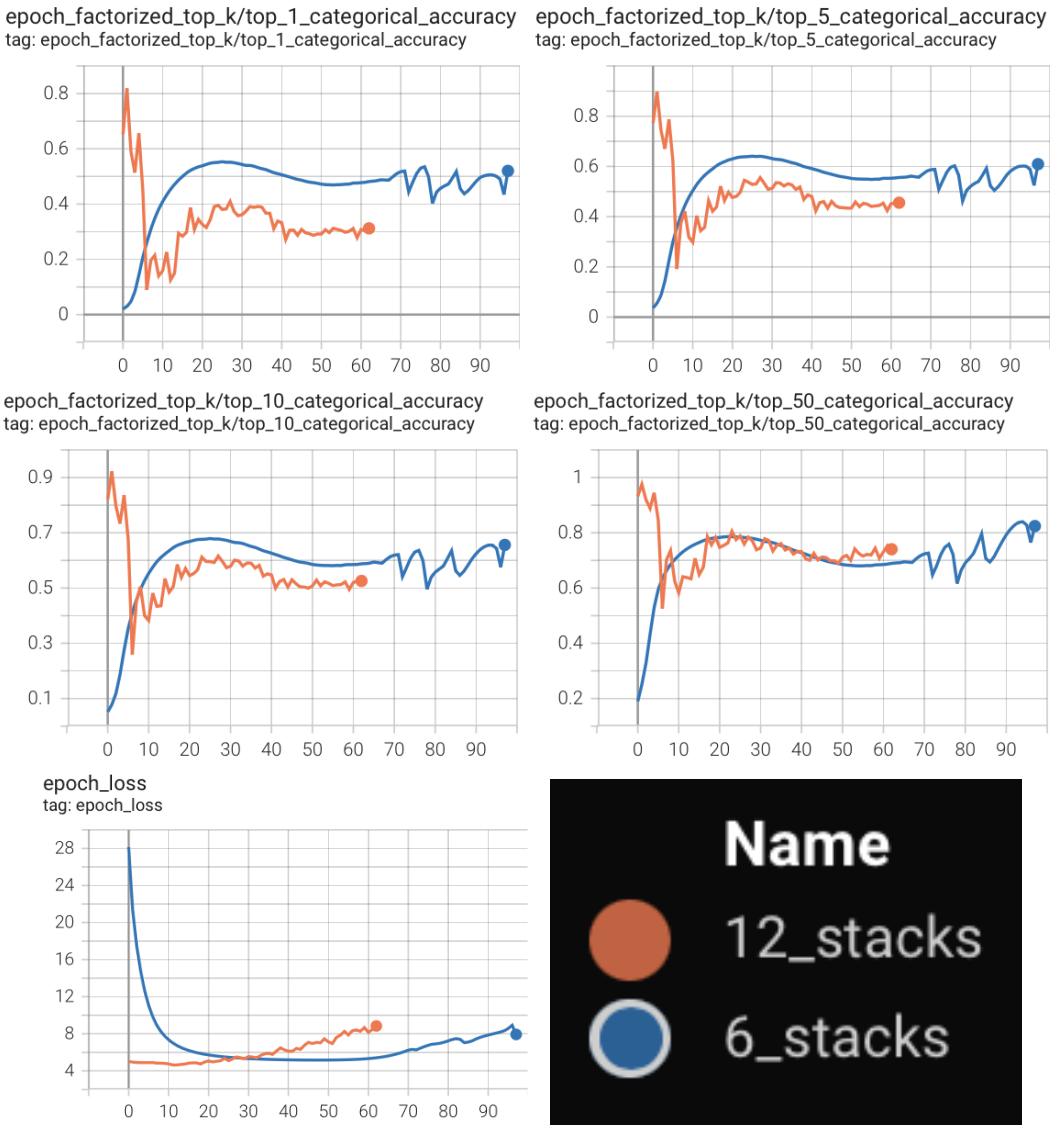


Figura 47: Accuracy TopK@10 y Loss para distintos stacks de atención en el encoder. Fuente: Elaboración Propia.

Attention Stack	Recall@10	Loss
12	0.6159	5.506
6	**0.6795**	**5.166**

Tabla 6: Tabla de Resultados Recall@10 y Categorical Cross Entropy

Comparacion Modelos

Por último, compararemos el mejor modelo de cada categoría de los benchmarks del Retrieval Stage, vamos a tomar como baseline el modelo base que implementa la libreria Tensorflow Recommenders

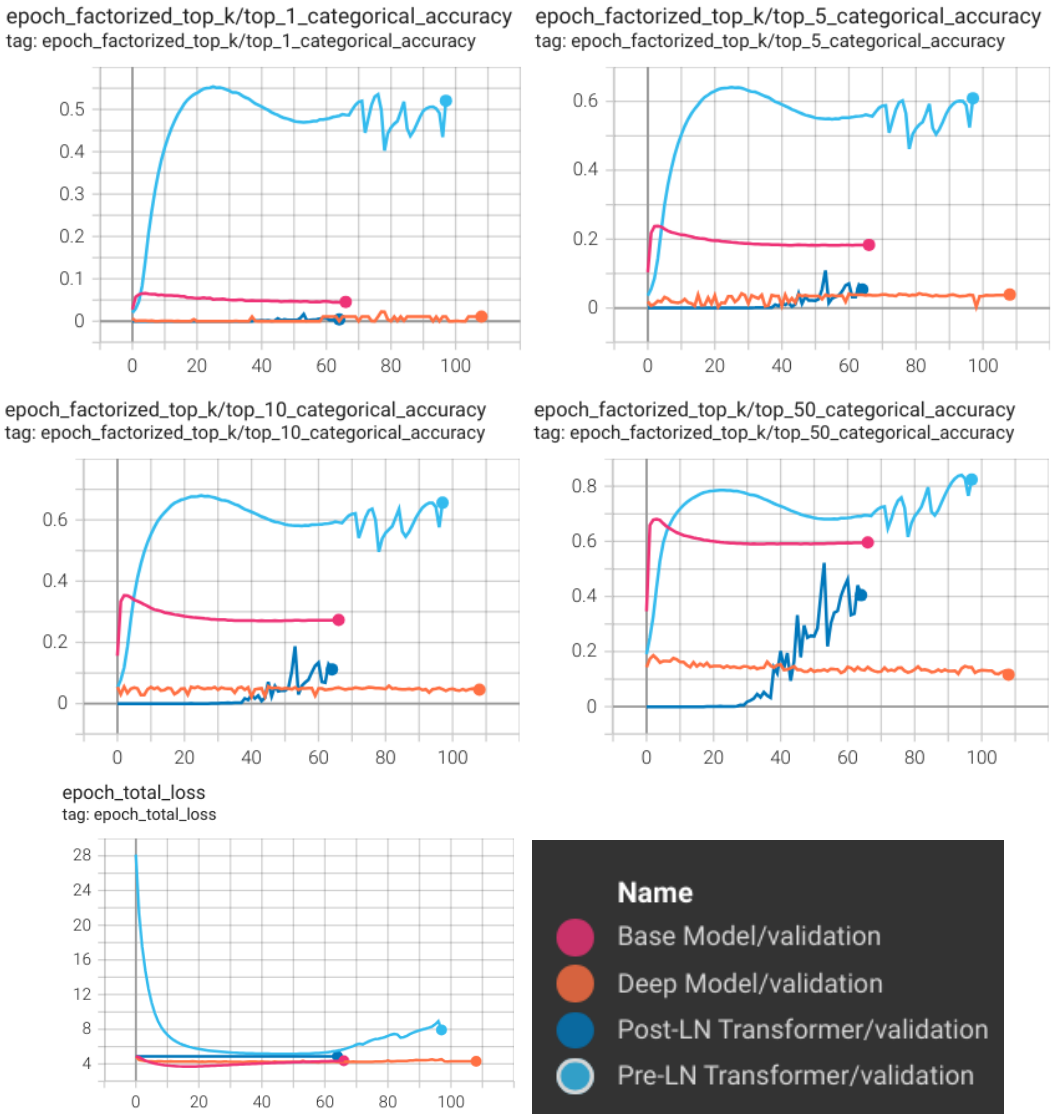


Figura 48: Accuracy TopK@10 y Loss para distintos Modelos del Retrieval Stage. Fuente: Elaboración Propia.

Modelo	Recall@10	Loss
Modelo Base (Baseline)	0.3529	3.715
Deep Model	0.0578	4.197
Post-LN Transformer (Vanilla)	0.112	4.875
Pre-LN Transformer	**0.6795**	**5.166**

Tabla 7: Tabla Comparativa de Modelos para el Retrieval Stage

4.3.2. Ranking Stage

Ahora evaluaremos la etapa de Ranking donde la funcion de perdida es **MSE**, pero también analizaremos otras metricas como MAE y RMSE.

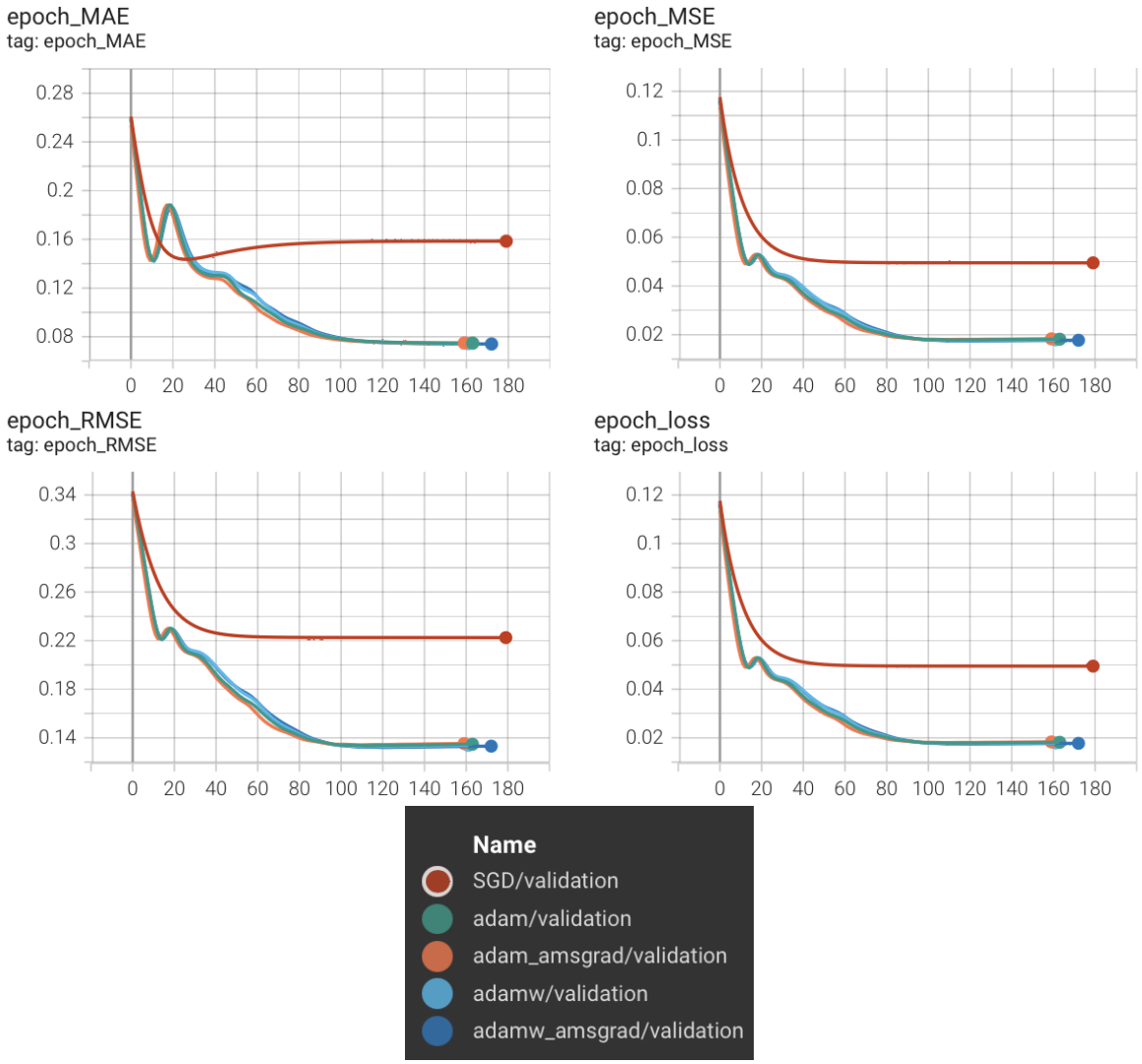


Figura 49: MAE, MSE y RMSE para distintos optimizadores.
Fuente: Elaboración Propia.

Optimizer	MAE	RMSE	Loss (MSE)
SGD	0.1436	0.2225	0.0495
Adam	0.0761	0.1335	0.0178
Adam+Amsgrad	0.0761	0.1341	0.0179
AdamW	0.0762	0.1327	0.0176
AdamW+Amsgrad	**0.075**	**0.1321**	**0.0174**

Tabla 8: Tabla de Resultados MAE, RMSE y MSE para distintos optimizadores.

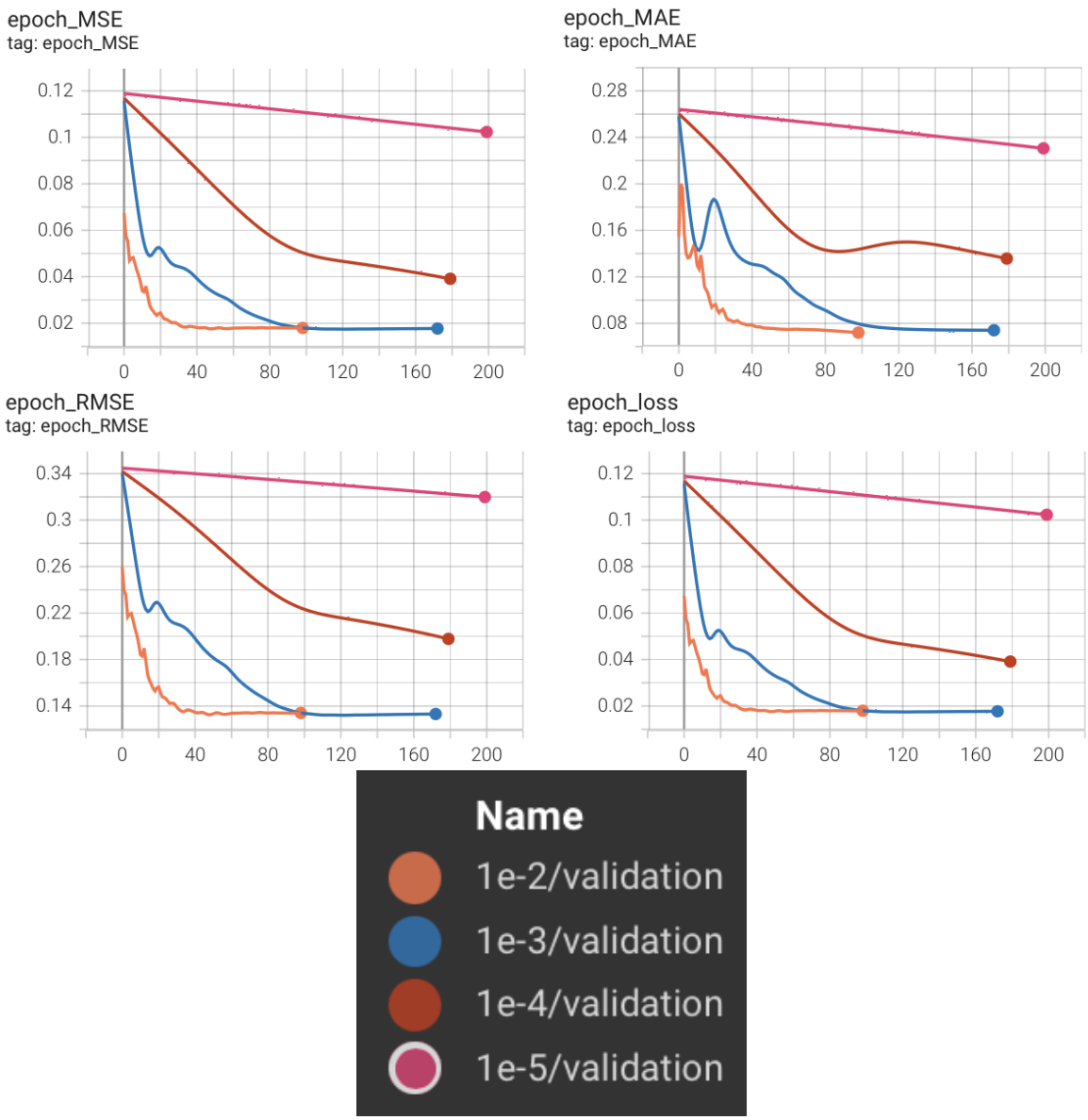


Figura 50: MAE, MSE y RMSE para distintos Learning Rates.
Fuente: Elaboración Propia.

Learning Rate	MAE	RMSE	Loss (MSE)
1e-2	0.0719	0.134	0.0179
1-3	**0.074**	**0.1321**	**0.0174**
1e-4	0.1357	0.1978	0.0391
1e-5	0.2305	0.3199	0.1023

Tabla 9: Tabla de Resultados MAE, RMSE y MSE para distintos Learning Rates.

Como se puede observar de los resultados, la implementación de la arquitectura **Pre-LN transformer** para el retrieval Stage fue la que obtuvo un mayor **accuracy** y **recall**, validando la solución planteada en esta memoria, el optimizador utilizado para esta arquitectura fue **AdamW** con **AMSGrad** con un learning rate de **1e-3** y con **6 stacks de encoders**.

Por otro lado, también se validó el modelo para el Ranking Stage con una **DCN** con el optimizador **AdamW** con **AMSGrad** y con un learning rate de **1e-3**.

4.4. Explicabilidad

La explicabilidad de los modelos de machine learning es un punto muy importante, pero a la vez es difícil poder comprender como funcionan por dentro estos modelos, sobre todo modelos de redes neuronales se los suele llamar *Cajas Negras* (Black Box), ya que cuentan con miles de pesos y parámetros.

Para el modelo de Retrieval se utilizó una arquitectura de transformers, esta arquitectura implementa métodos de atención y vamos a visualizar las matrices de cada cabezal atencional como un método de explicabilidad para el Query y Candidate Model.

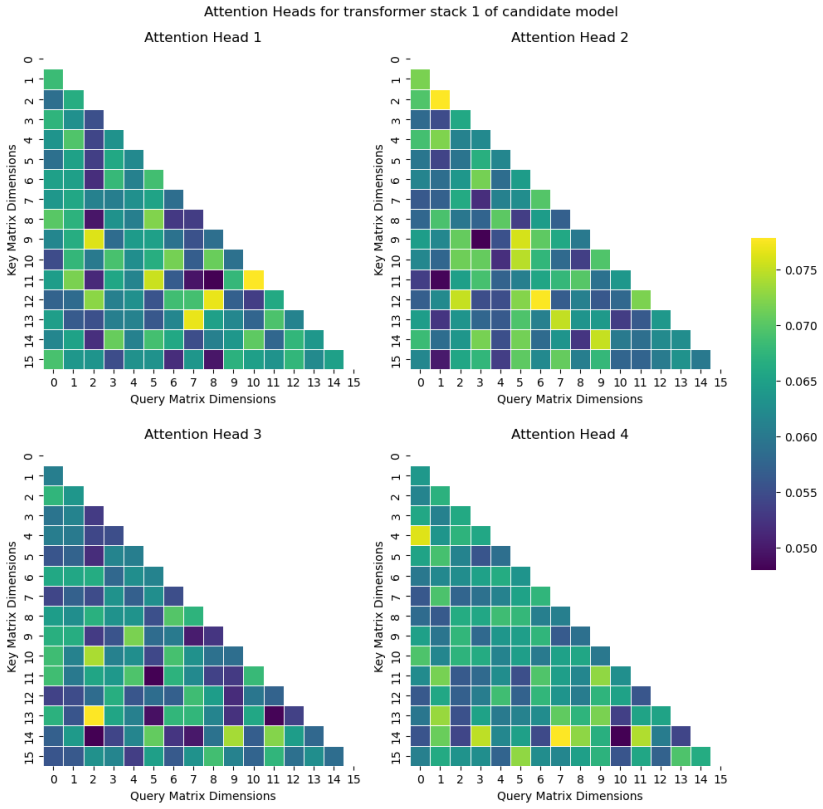


Figura 51: Cabezales Atencionales Primer Stack Candidate Model.
Fuente: Propia.

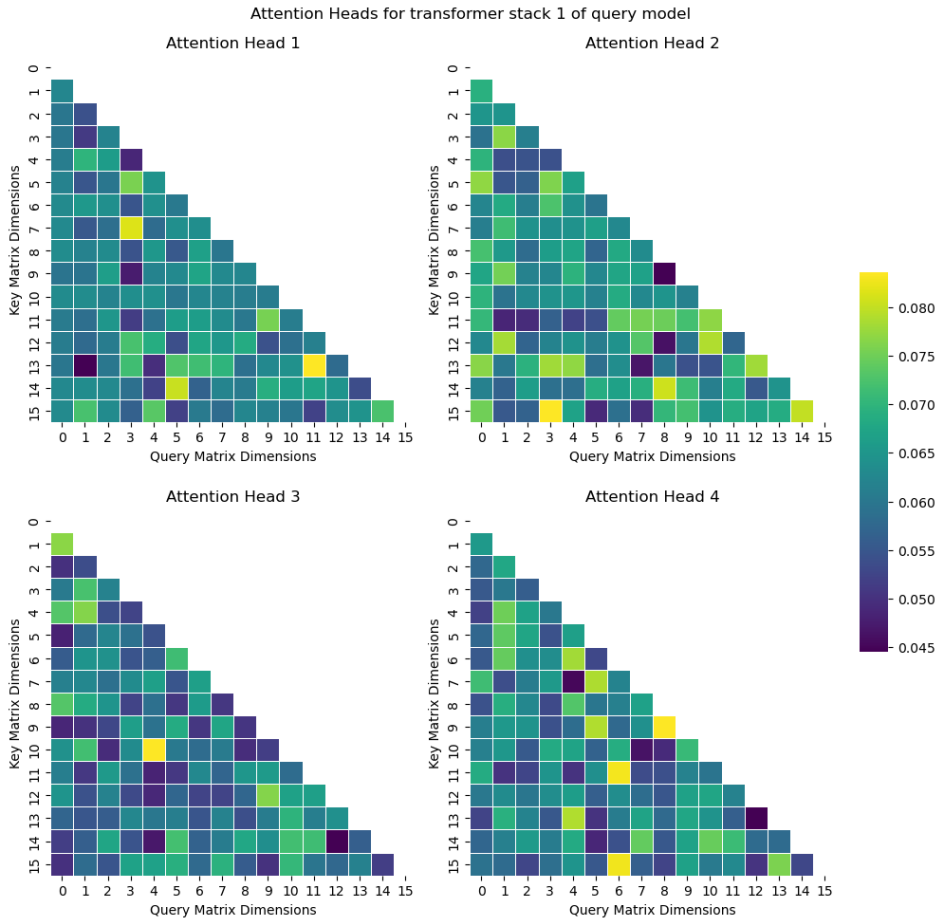


Figura 52: Cabezales Atencionales Primer Stack Query Model.
Fuente: Propia.

Podemos observar como interactúan las distintas dimensiones de las matrices Query y Key de los cabezales de atención, esto es para el primer stack que recibe los embeddings de los usuarios y los productos, en el anexo se encuentran las matrices para los otros stacks del transformer, pero tampoco nos vamos a centrar mucho en este modelo, ya que solo recibe embeddings de usuarios y productos, además que hay un debate sobre la explicabilidad de los transformes y algunos investigadores plantean utilizar métodos de prominencia (Saliency Methods) en vez de atención [Bastings y Filippova, 2020], pero esto queda a criterio personal, lo que sería interesante para trabajos futuros sería poder analizar distintos métodos de explicabilidad y visualizar el espacio de embeddings aprendido por la red y como se mapean los distintos vectores multidimensionales, mediante el uso de PCA y T-SNE.

Para el modelo de Ranking podemos analizar los pesos de la DCN y ver como la red aprende el cruce, las distintas características. La matriz de pesos W de la DCN revela que cruce de características el modelo ha aprendido que son importantes, cada feature está representada por un embedding de dimensión 16, por lo tanto, la importancia estará caracterizada por

él (i, j)-ésimo bloque W_{ij} que es de dimensión 16 por 16. A continuación, visualizamos la norma de Frobenius $||W_{ij}||_F$ de cada bloque, y una norma más grande sugeriría una mayor importancia (suponiendo que las incorporaciones de las características sean de escalas similares).

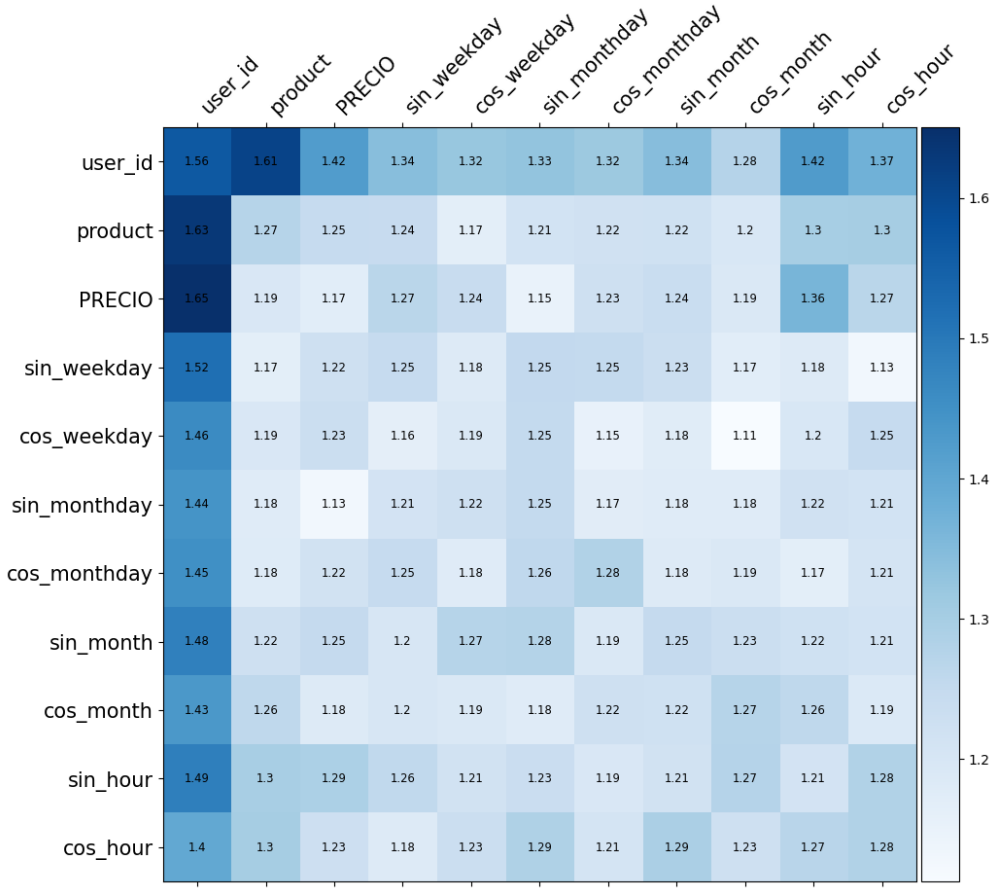


Figura 53: Matriz de pesos W de la DCN.
Fuente: Propia.

Además de la norma de bloque, también podríamos visualizar la matriz completa, o el valor medio/mediano/máximo de cada bloque. Aquí podemos ver como la red aprende que el embedding del usuario se correlaciona bastante con el embedding del producto y de su precio. También hay algunas relaciones no tan fuertes, pero igual pueden ser importantes como el precio con el seno de la hora. Hay que tener en cuenta que los embeddings tienen alta dimensionalidad y a la vez son 11 features, esto nos da un vector de features de $16 \cdot 11 = 176$, un vector de 176 dimensiones es difícil de explicar y la matriz de pesos es uno de los métodos más cercanos que tenemos para comprender al modelo. Queda como trabajo futuro implementar algún otro método de explicabilidad como lo puede ser LIME [Ribeiro *et al.*, 2016], SHAP[Lundberg y Lee, 2017] o BertViz [Vig, 2019] para el caso del transformer.

4.5. Despliegue a Producción

Para desplegar el modelo a producción se utilizó **Docker**, las imágenes utilizadas fueron desarrolladas por Google y diseñadas específicamente para desplegar soluciones de machine learning. Para el modelo de Retrieval se utilizó **ScaNN**.

Listing 1 Contenedor Docker para el Modelo de Retrieval

```
#!/bin/bash
docker run -p -d --name recsys_retrieval 8501:8501 \
  --mount type=bind,source=/models/scann/model_name,target=/models/retrieval \
  -e MODEL_NAME=retrieval -t google/tf-serving-scann
```

Para el modelo de Ranking se utilizó **TensorFlow Serving**

Listing 2 Contenedor Docker para el Modelo de Ranking

```
#!/bin/bash
docker run -p -d --name recsys_ranking 8501:8501 \
  --mount type=bind,source=/models/ranking/model_name,target=/models/ranking \
  -e MODEL_NAME=ranking -t tensorflow/serving
```

Luego, para hacer consultas a los modelos, se utilizan peticiones HTTP POST mediante el uso de JSON

Listing 3 Petición POST para modelo de Retrieval

```
#!/bin/bash
POST http://localhost:8501/v1/models/retrieval:predict
Content-Type: application/json

{
  "instances": [
    "User_id"
  ]
}
```

También en el repositorio de GitHub del proyecto están disponibles scripts de Python para hacer las consultas de manera automatizada a los modelos, permitiendo hacer consultas de manera más cómoda, pero se puede utilizar cualquier lenguaje de programación para consultar los modelos, de hecho Cropy está desarrollado en **Ruby on Rails** y será el lenguaje con el que se consultara al modelo en un futuro.

Listing 4 Petición POST para modelo de Ranking

```
#!/bin/bash
POST http://localhost:8501/v1/models/ranking:predict
Content-Type: application/json

{
  "instances": [
    "User_id",
    "product",
    "PRECIO",
    "sin_weekday",
    "cos_weekday",
    "sin_monthday",
    "cos_monthday",
    "sin_month",
    "cos_month",
    "sin_hour",
    "cos_hour",
  ]
}
```

Listing 5 Uso del Script de Python para el modelo de Retrieval

```
#!/bin/bash
python predict_retrieval.py -i User_id -k 10
```

Donde -u es el id del usuario y -k es el número de productos a recomendar.

Listing 6 Uso del Script de Python para el modelo de Ranking

```
#!/bin/bash
python predict_ranking.py -u User_id -f product, PRECIO, \
sin_weekday, cos_weekday, sin_monthday, cos_monthday, \
sin_month, cos_month, sin_hour, cos_hour
```

Donde -u es el id del usuario y -f son las features del producto a rankear.

De esta manera se pueden desplegar los modelos a producción y aislarlos mediante contenedores Docker, permitiendo poder ejecutar las predicciones desde cualquier lugar y dispositivo de manera fácil y sencilla, un objetivo esencial para Cropy. Todas las instrucciones, desde como entrenar el modelo, el formato del dataset y como desplegar el sistema recomendador se encuentra explicado en el GitHub del proyecto (ver sección 3.6), esto para que otras personas puedan utilizar la arquitectura o reproducir los resultados de esta memoria.

4.6. Complejidad y Rendimiento

Para que el sistema recomendador sea una herramienta útil en el día a día, necesitamos evaluar su rendimiento y complejidad, ya que de esto depende si el modelo podrá aguantar las peticiones, si podrá generar las recomendaciones en un tiempo aceptable para el usuario y si su tamaño le permite ser desplegado en la nube.

4.6.1. Parámetros Entrenables

Una forma de medir la complejidad del sistema recomendador es analizando la cantidad de parámetros entrenables que posee, para la etapa de Retrieval se utilizó un enfoque *Two Tower* donde cada torre está conformada por 6 stacks de encoders, donde para la dimensión del embedding (determinado por (4)) se tomó el máximo entre la cantidad de usuarios únicos y la cantidad productos únicos, resultando en un $d_{model} = 16$. Los parámetros entrenables de cada encoder se pueden calcular de la siguiente manera:

- Attention Layer: $d_{model} \cdot d_{model} * 3 = 768$.
- Linear Layer: $d_{model} \cdot d_{model} + d_{model} = 272$.
- FeedForward Layer In: $d_{model} \cdot (4 \cdot d_{model}) + (4 \cdot d_{model}) = 1088$.
- FeedForward Layer Out: $(4 \cdot d_{model}) \cdot d_{model} + d_{model} = 1040$.

Sumando todo nos da que cada encoder tiene **3168** parámetros entrenables, como tenemos 6 stacks de encoders por cada torre, esto nos da un total de **19008** parámetros entrenables por torre, por lo que en la etapa de retrieval tenemos **38016** parámetros entrenables entre las dos torres.

Para la etapa de Ranking se utilizó una DCN y capas densas, donde L_{size} es el tamaño de la hidden layer, N_{layers} es la cantidad de capas de la hidden layer. El modelo de ranking recibe como entrada un vector compuesto por las 9 features más los dos embeddings de usuarios y productos, dando un total 11 features, cada feature la proyectamos con un embedding de tamaño 16, dando un total de $d_{model} = 16 \cdot 11 = 176$, con $N_{layers} = 4$, $L_{size} = 32$ y $n_{features} = 11$ los parámetros entrenables de la etapa de ranking se pueden calcular de la siguiente manera:

- DCN: $d_{model} \cdot d_{model} + d_{model} = 31152$.
- Hidden Layers In: $d_{model} \cdot L_{size} + L_{size} = 5664$.
- Hidden Layers Mid: $(L_{size} \cdot L_{size} + L_{size}) \cdot N_{layers} - 1 = 1056 \cdot 3 = 3168$.
- Hidden Layers Out: $L_{size} + 1 = 33$.

Sumando todo, nos da un total de, **40017** parámetros entrenables.

Ambas etapas tienen aproximadamente la misma cantidad de parámetros y podríamos decir que en total todo el sistema recomendador tiene, **78033** parámetros entrenables, lo que no es tanto hoy en día cuando las redes neuronales tienen millones de parámetros entrenables.

4.6.2. Entrenamiento y Tamaño de los Modelos

Para la etapa de Retrieval, se intentó ajustar el batch size al mayor tamaño posible, ya que de otra forma el entrenamiento demoraba mucho tiempo, al final se utilizó un **batch size de 4096**, ocupando un total de **6824 MiB memoria RAM** de la GPU, el modelo se demoró en promedio **28 segundos por epoch** y el total del entrenamiento fue de aproximadamente de 2800 segundos o 45 minutos.

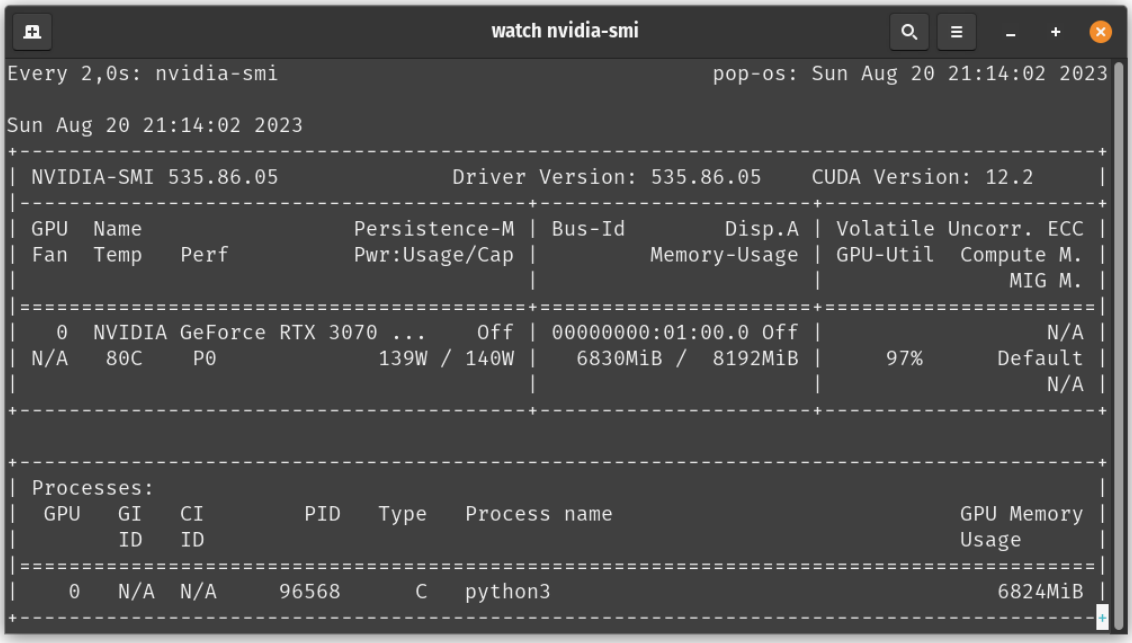


Figura 54: Estadísticas de la GPU durante el entrenamiento del Retrieval Stage.
Fuente: Propia.

Para la etapa de Ranking, se utilizó un batch size igual al la cantidad de datos de entrenamiento, así todo el dataset fue insertado dentro del batch size, esto fue gracias a que la DCN y las hidden layers no ocupan tanto espacio en la GPU, ocupando un total de **4668 MiB de memoria RAM** de la GPU con el dataset completo cargado en memoria, el modelo se demoró aproximadamente **0.88 segundos por epoch** y el tiempo total aproximado del entrenamiento fue de aproximadamente 180 segundos o 3 minutos, ya que se utilizó early stopping.

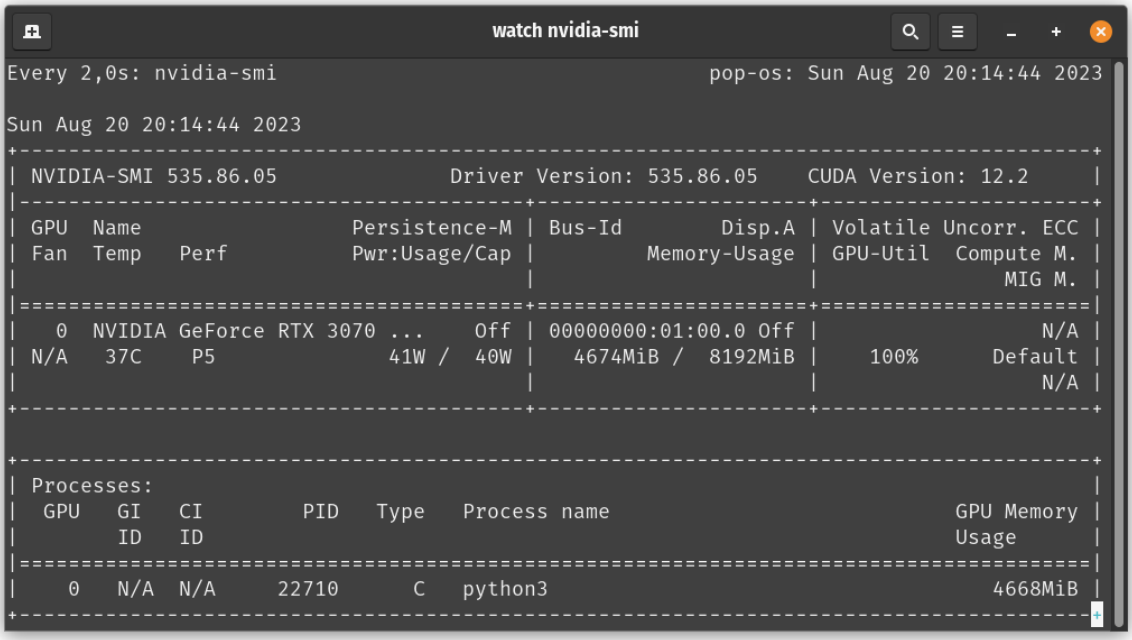


Figura 55: Estadísticas de la GPU durante el entrenamiento del Ranking Stage.
Fuente: Propia.

Figura 56: Tamaño del modelo de Retrieval y Ranking.
Fuente: Elaboración Propia.

Para el modelo de Retrieval, se utilizó el formato de ScaNN, por lo que el modelo final guardado tiene un tamaño mucho menor, de aproximadamente **8,4 MB**, en cambio, para el modelo de Ranking se utilizó el formato universal de TensorFlow y Keras, sin aplicar optimizaciones el modelo tiene un tamaño aproximadamente de **39 MB**, ambos modelos están dentro del tamaño aceptable por lo que se cumple con los objetivos de la memoria.

4.6.3. Inferencia

Una vez desplegado nuestros modelos, se realizó un benchmark para analizar el rendimiento de la inferencia del sistema recomendador, se realizó una recomendación de 5 productos, y se repitió durante 10000 iteraciones, para saturar al modelo y ver que tal se comporta bajo alta demanda.

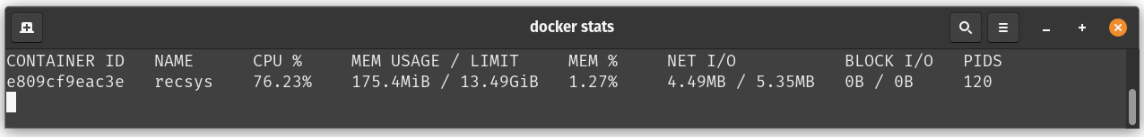


Figura 57: Estadísticas del contenedor Docker del Sistema Recomendador.
Fuente: Propia.

Como se puede observar de las estadísticas de Docker, el contenedor utiliza aproximadamente **75 % del CPU**, **175 MiB de uso de RAM** y **5 MB de uso de Red**, hay que tener en cuenta que el uso de CPU es alto, ya que estamos realizando muchas peticiones a la vez, esto no necesariamente representa el uso real, es un test exagerado para asegurar de que el sistema recomendador pueda aguantar bajo alta carga.

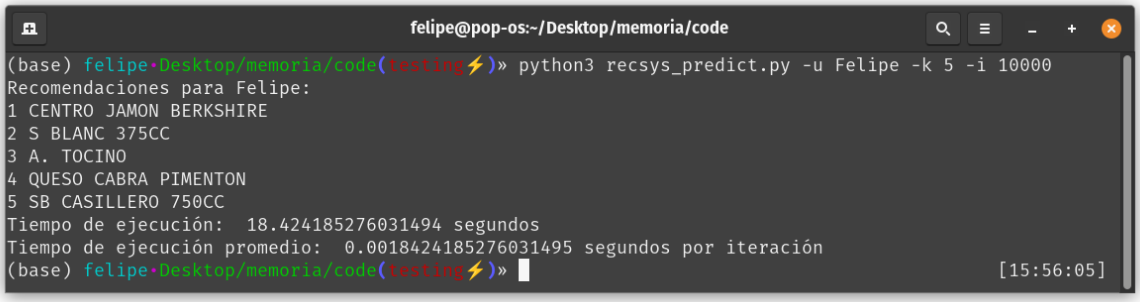


Figura 58: Tiempos de Inferencia para El sistema Recomendador.
Fuente: Propia.

Acá podemos observar el funcionamiento del script de Python para el sistema recomendador, se realizaron 10000 iteraciones y se retornó el último llamado a la API, para **10.000 peticiones** el tiempo de inferencia total fue de aproximadamente **19 segundos**, lo que nos deja un total de aproximadamente **0.0019 segundos por iteración**, esto significa que podríamos realizar **526 peticiones en 1 segundo**, lo que es un excelente tiempo de iteración y demuestra lo eficiente que es el sistema recomendador y la librería ScaNN, cumpliendo con los requerimientos planteados en esta memoria.

CAPÍTULO 5
CONCLUSIONES

En conclusión, el modelo recomendador propuesto se presenta como una solución innovadora y efectiva para mejorar la experiencia de los usuarios en bares y restaurantes. A través del uso de redes neuronales y la arquitectura de Transformers, se logra un mayor accuracy en las recomendaciones, cumpliendo con los objetivos planteados de esta memoria, ya que cuenta con tiempos aceptables de inferencia, presenta distintos métodos de explicabilidad de los modelos del sistema recomendador, cuenta con una pipeline automatizada de entrenamiento e inferencia mediante contenedores Docker que son fácilmente escalables. se espera que esta propuesta pueda ser una fuente de inspiración para futuros trabajos en el ámbito de los sistemas de recomendación, y que pueda ser adaptada y mejorada para su aplicación en otros contextos y sectores. Es importante destacar que el modelo recomendador propuesto tiene un gran potencial para ser aplicado en otros ámbitos, más allá de los bares y restaurantes. Por ejemplo, podría ser utilizado en tiendas en línea para recomendar productos a los usuarios, en servicios de streaming para sugerir contenido audiovisual, o incluso en el ámbito de la salud para recomendar tratamientos médicos personalizados. En cuanto a las contribuciones y aplicaciones del trabajo realizado, es importante destacar que esta propuesta no solo beneficia a los usuarios y a los establecimientos gastronómicos, sino que también tiene un impacto positivo en la industria tecnológica. El uso de redes neuronales y Transformers en sistemas de recomendación es una técnica innovadora y prometedora que puede ser aplicada en una amplia variedad de contextos, lo que abre nuevas oportunidades de investigación y desarrollo en este campo. En resumen, la propuesta de solución presentada es una muestra del potencial de la tecnología para mejorar la calidad de vida de las personas y optimizar los procesos en distintos ámbitos. A través de su trabajo, se demuestra que la combinación de la inteligencia artificial y la gastronomía puede generar soluciones innovadoras y efectivas que pueden ser aplicadas en otros contextos y sectores.

5.1. Trabajo Futuro

Posibles trabajos futuros y áreas de mejora para el sistema recomendador desarrollado son:

- Implementación de Flash Attention [Dao *et al.*, 2022] [Dao, 2023] para mejorar la eficiencia del Transformer.
- Implementación de más métodos de XAI como: BertViz, SHAP, LIME, Saliency Methods, etc.
- Reemplazar la arquitectura del Transformer por Backpacks [Hewitt *et al.*, 2023] que cuentan con mayor interpretabilidad.
- Implementar métodos de Cuantización de bytes [Dettmers *et al.*, 2022] y/o precisión mixta [Micikevicius *et al.*, 2018] para optimizar el Transformer.

ANEXOS

Listing 7 Código para calcular UD usado en (1)

```
    user_dict = {}
    user_count = {}
    for user, product in zip(df['user_id'], df['product']):
        if user not in user_count:
            user_count[user] = {}
        if product in user_count[user]:
            user_count[user][product] += 1
        else:
            user_count[user][product] = 1
    user_dict = copy.deepcopy(user_count)
    for user in user_dict:
        total = sum(user_dict[user].values())
        for product in user_dict[user]:
            user_dict[user][product] /= total
```

Listing 8 Código para calcular IUD e ITD usado en (1)

```
    store_df = df.groupby('store')[['user_id']].agg(['count', 'nunique'])
    store_dict = store_df.to_dict('index')
    item_total_dict = {}
    item_unique_dict = {}
    item_helper = {}
    for user, product, local in zip(df['user_id'], df['product'], df['nomcc']):
        if product not in item_helper:
            item_total_dict[product] = 0
            item_unique_dict[product] = 0
            item_helper[product]['user_list'] = []
        if user not in item_helper[product]['user_list']:
            item_helper[product]['user_list'].append(user)
            item_unique_dict[product] += 1/store_dict[local]['nunique']
        item_total_dict[product] += 1/store_dict[local]['count']
```

Listing 9 Código para calcular el Ranking formulado en (1)

```
    df['ranking'] = df.apply(lambda x:
    ↪  user_dict[x['user_id']][x['product']] + 0.1 *
    ↪  (1-1/len(user_dict[x['user_id']])) *
    ↪  np.random.uniform(item_total_dict[x['product']],
    ↪  1-item_unique_dict[x['product']]), axis=1)
```

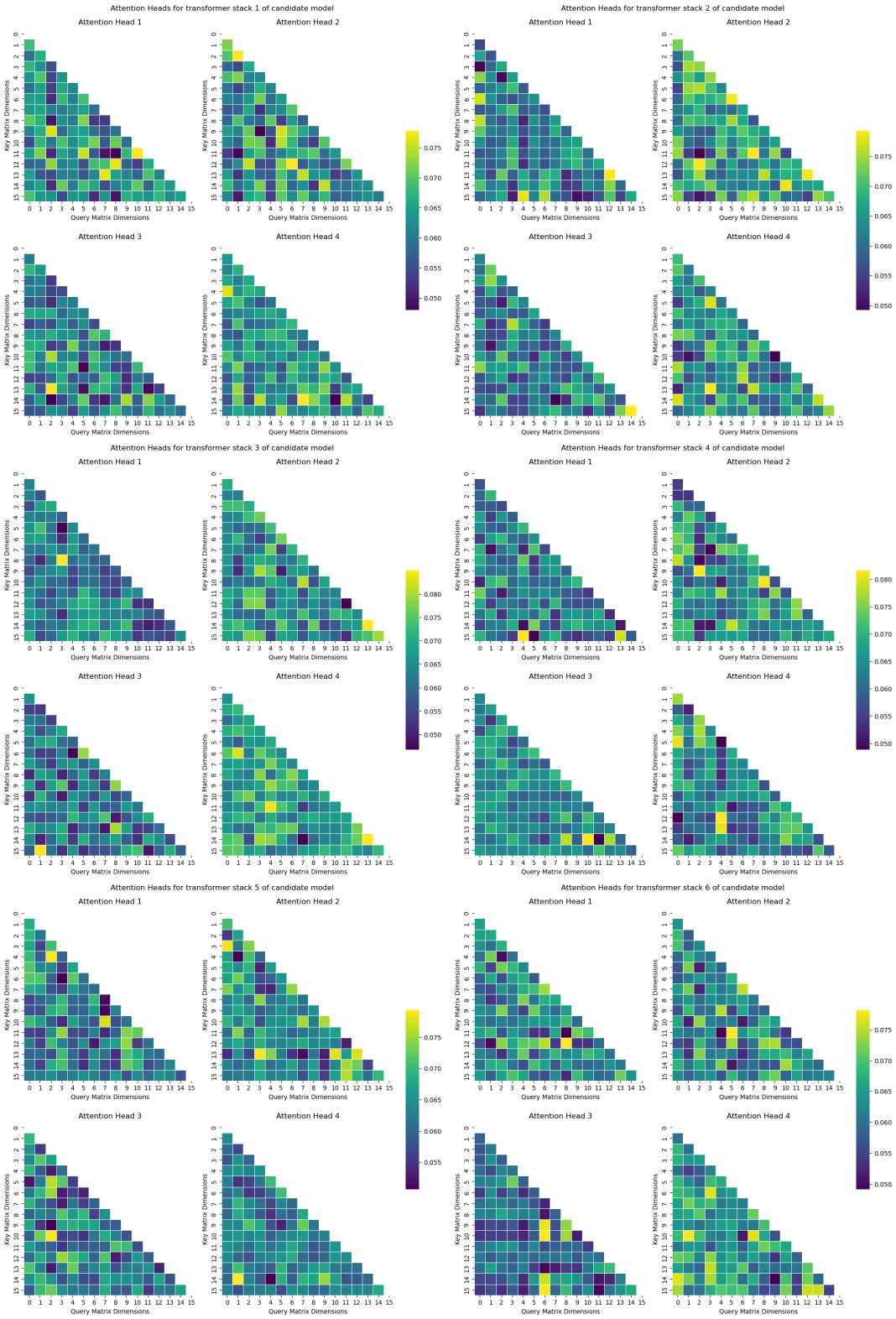


Figura 59: Cabezales Atencionales Candidate Model.
Fuente: Elaboración Propia.

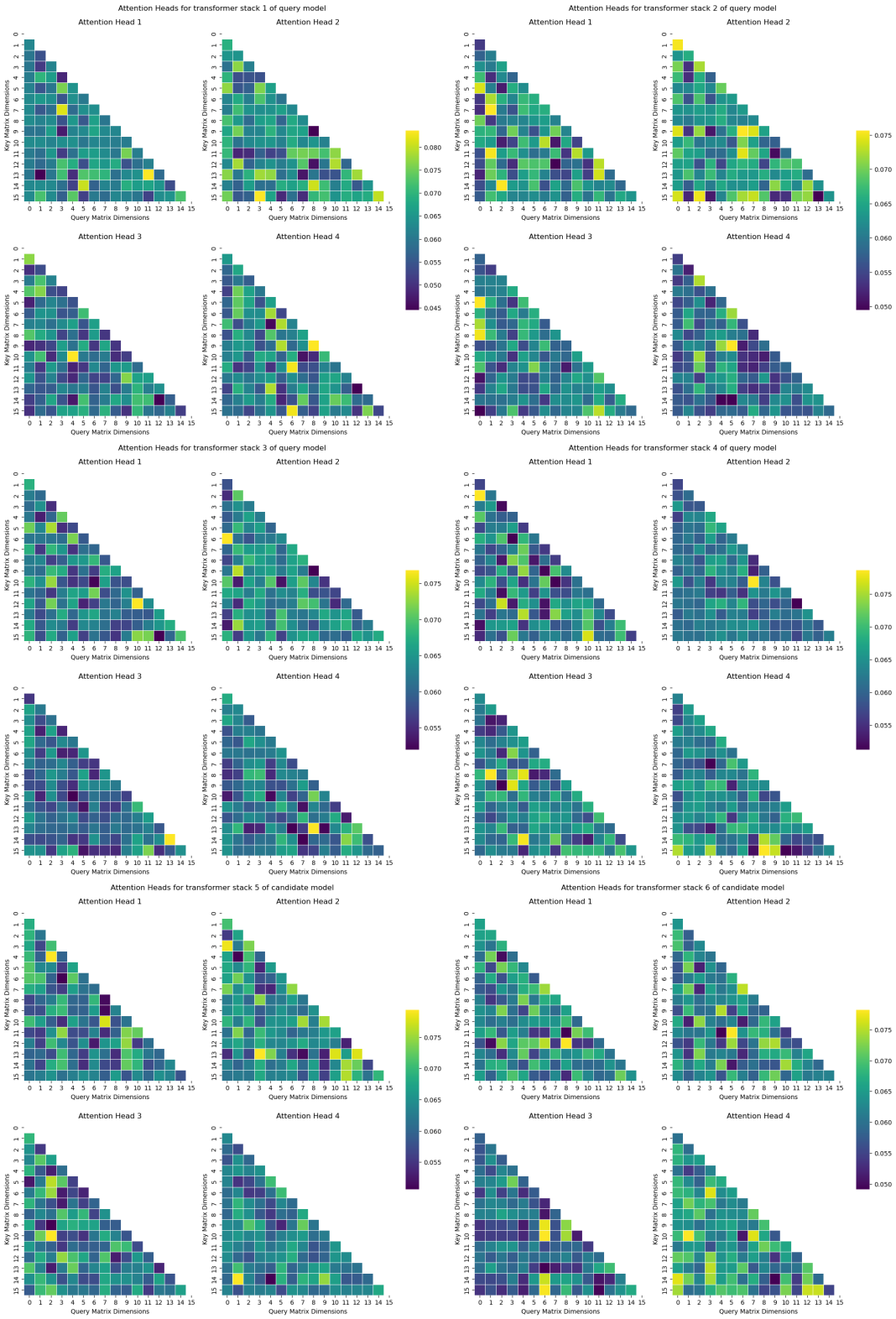


Figura 60: Cabezales Atencionales Query Model.
Fuente: Elaboración Propia.

REFERENCIAS BIBLIOGRÁFICAS

[Aggarwal, 2016] Aggarwal, C. C. (2016). *Recommender Systems*. Springer International Publishing.

[Ba *et al.*, 2016] Ba, J. L., Kiros, J. R., y Hinton, G. E. (2016). Layer normalization.

[Bahdanau *et al.*, 2016] Bahdanau, D., Cho, K., y Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate.

[Bastings y Filippova, 2020] Bastings, J. y Filippova, K. (2020). The elephant in the interpretability room: Why use attention as explanation when we have saliency methods? En *Proceedings of the Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 149–155, Online. Association for Computational Linguistics.

[Beel *et al.*, 2013] Beel, J., Langer, S., Nürnberger, A., y Genzmehr, M. (2013). The impact of demographics (age and gender) and other user-characteristics on evaluating recommender systems. En *International Conference on Theory and Practice of Digital Libraries*, pp. 396–400. Springer.

[Bennett y Lanning, 2007] Bennett, J. y Lanning, S. (2007). The netflix prize. En *Proceedings of KDD cup and workshop*, volumen 2007, p. 35. New York.

[Dao, 2023] Dao, T. (2023). FlashAttention-2: Faster attention with better parallelism and work partitioning.

[Dao *et al.*, 2022] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., y Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. En *Advances in Neural Information Processing Systems*.

[Datta *et al.*, 2023] Datta, S., Roy, M., y Kar, P. (2023). *Recommender Systems*. CRC Press.

[Dettmers *et al.*, 2022] Dettmers, T., Lewis, M., Belkada, Y., y Zettlemoyer, L. (2022). Llm.int8(): 8-bit matrix multiplication for transformers at scale.

[Devlin *et al.*, 2019] Devlin, J., Chang, M.-W., Lee, K., y Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

[Dosovitskiy *et al.*, 2021] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., y Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale.

[Glorot y Bengio, 2010] Glorot, X. y Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. En Teh, Y. W. y Titterington, M., editores, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volumen 9 de *Proceedings of Machine Learning Research*, pp. 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[Goodfellow *et al.*, 2016] Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press.

[He *et al.*, 2015a] He, K., Zhang, X., Ren, S., y Sun, J. (2015a). Deep residual learning for image recognition.

[He *et al.*, 2015b] He, K., Zhang, X., Ren, S., y Sun, J. (2015b). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.

[Hendrycks y Gimpel, 2020] Hendrycks, D. y Gimpel, K. (2020). Gaussian error linear units (gelus).

[Hewitt *et al.*, 2023] Hewitt, J., Thickstun, J., Manning, C. D., y Liang, P. (2023). Backpack language models.

[Jannach *et al.*, 2010] Jannach, D., Zanker, M., Felfernig, A., y Friedrich, G. (2010). *Recommender Systems*. Cambridge University Press.

[Jouppi *et al.*, 2023] Jouppi, N. P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z., y Patterson, D. (2023). Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings.

[Kaggle, 2022] Kaggle (2022). 2022 kaggle machine learning and data science survey. `https://www.kaggle.com/competitions/kaggle-survey-2022/`. Accessed: 2023-01-17.

[Kumar *et al.*, 2021] Kumar, P., Vairachilai, S., Potluri, S., y Mohanty, S. (2021). *Recommender Systems: Algorithms and Applications*. CRC Press.

[Lin *et al.*, 2021] Lin, T., Wang, Y., Liu, X., y Qiu, X. (2021). A survey of transformers.

[Loshchilov y Hutter, 2019] Loshchilov, I. y Hutter, F. (2019). Decoupled weight decay regularization.

[Lundberg y Lee, 2017] Lundberg, S. y Lee, S.-I. (2017). A unified approach to interpreting model predictions.

[M, 2021] M, G. (2021). Balance de la industria gastronómica: 46 % de pérdidas de empleo y caída en las ventas sobre el 50 %.

[Micikevicius *et al.*, 2018] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., y Wu, H. (2018). Mixed precision training.

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill.

[Murphy, 2012] Murphy, K. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning series. MIT Press.

[OpenAI, 2023] OpenAI (2023). Gpt-4 technical report.

[Popel y Bojar, 2018] Popel, M. y Bojar, O. (2018). Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70.

[Radford *et al.*, 2018] Radford, A., Narasimhan, K., Salimans, T., y Sutskever, I. (2018). Improving language understanding by generative pre-training.

[Reddi *et al.*, 2018] Reddi, S. J., Kale, S., y Kumar, S. (2018). On the convergence of adam and beyond. En *International Conference on Learning Representations*.

[Ribeiro *et al.*, 2016] Ribeiro, M. T., Singh, S., y Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier.

[Ricci *et al.*, 2011] Ricci, F., Rokach, L., Shapira, B., y Kantor, P. B., editores (2011). *Recommender Systems Handbook*. Springer US.

[Saravia, 2021] Saravia, C. (2021). Actividad del sector gastronómico todavía no vuelve a los niveles previos a la pandemia.

[Srivastava *et al.*, 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

[StackOverflow, 2022] StackOverflow (2022). Stack overflow developer survey 2022. `https://survey.stackoverflow.co/2022/`. Accessed: 2023-01-17.

[Tay *et al.*, 2022] Tay, Y., Dehghani, M., Bahri, D., y Metzler, D. (2022). Efficient transformers: A survey.

[Vaswani *et al.*, 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., y Polosukhin, I. (2017). Attention is all you need.

[Vig, 2019] Vig, J. (2019). A multiscale visualization of attention in the transformer model. En *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 37–42, Florence, Italy. Association for Computational Linguistics.

[Wang *et al.*, 2017] Wang, R., Fu, B., Fu, G., y Wang, M. (2017). Deep & cross network for ad click predictions.

[Wang *et al.*, 2021] Wang, R., Shivanna, R., Cheng, D., Jain, S., Lin, D., Hong, L., y Chi, E. (2021). Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. En *Proceedings of the web conference 2021*, pp. 1785–1797.

[Xiong *et al.*, 2020] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., y Liu, T.-Y. (2020). On layer normalization in the transformer architecture.