Facultad
de
Ciencias

NO TODO LO QUE NECESITAS ES ATENCIÓN
(Attention is not all you need)

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Diego Rivera López-Brea

Director: Cristina Tirnauca

Septiembre - 2022

"Cum hoc ergo propter hoc."

Falacia de la causa falsa

Agradecimientos

A mi familia y amigos, por el apoyo incondicional.

A Mariela, por animarme a terminar.

A Carlota y Miguel por darme la oportunidad de hacerlo.

A Cristina por su dirección y guía.

Resumen

En este trabajo de Fin de Grado se explora el mundo del Procesamiento del Lenguaje Natural y el uso de redes neuronales para este campo específico de la Inteligencia Artificial. Se hace especial incidencia en la arquitectura *transformer*, propuesta en un primer momento en el ahora famoso artículo *Attention is all you need* y dominante en el sector desde entonces. Pese a sus éxitos, los *transformers* tienen sus detractores y a lo largo del trabajo se explican y se ejemplifican algunas de las críticas que reciben, implementándolos en un entorno de producción real y en un problema práctico de búsqueda y recuperación de información. A su vez, se presenta una línea de investigación muy reciente que pretende dotar a estos modelos de lo que carecen: sentido común. Esto es, la inferencia causal y el razonamiento contrafactual como posible pieza clave en la descripción formal de los lenguajes naturales.

Palabras clave: Aprendizaje Automático Profundo, *tranformers*, *BERT*, inferencia causal, Procesamiento del Lenguaje Natural.

Abstract

In this Final Degree Project, the world of Natural Language Processing and the use of neural networks for this specific field of Artificial Intelligence are explored. Special emphasis is placed on the *transformer* architecture, initially proposed in the now famous article *Attention is all you need* and dominant in the sector ever since. Despite their successes, the *transformers* have their detractors, and throughout this work, some of the criticisms they receive are explained and exemplified, implementing them in a real production environment and in a practical information search and retrieval problem. At the same time, a very recent line of research is presented that aims to provide these models with what they lack: common sense. That is, causal inference and counterfactual reasoning as a possible key piece in the formal description of natural languages.

Keywords: Deep Learning, transformers, BERT, causal inference, Natural Language Processing.

Contenidos

Capítulo 1

Introducción

En 2017 el mundo del Procesamiento de Lenguaje Natural, (NLP) por sus siglas en inglés, se vio sacudido por un artículo publicado por un equipo de Google. En él, se introducía una nueva arquitectura de red neuronal conocida como *transformer*, que revolucionaría este campo de la Inteligencia Artificial hasta el día de hoy. Su título, buque insignia del actual estado del arte en este sector, lanzaba una poderosa aseveración: *Attention is all you need*, "Todo lo que necesitas es atención"[1].

Los lenguajes naturales son los lenguajes del mundo. Desde un punto de vista histórico, el desarrollo de las lenguas se debe al desarrollo de las sociedades, a las migraciones y a la mezcla cultural. Hay quien defiende que en un inicio el lenguaje respondía a la necesidad de comunicación. Hay también quien defiende que responde antes al impulso de imposición. En cualquier caso, lo que parece estar claro es que en su desarrollo nunca hubo un consenso que asegurase su falta de ambigüedad.

Es por esto que dar una definición de matemáticas utilizando un lenguaje natural resulta ser una tarea complicada. La paradoja consiste en utilizar un lenguaje no optimizado para definir, pero que responde a las leyes de la inferencia que determinan nuestro entendimiento, precisamente cuando la definición de matemáticas, si es que la hubiera, pasaría por decir que, en gran medida, es el estudio de las leyes de la inferencia y de su representación lingüística.

Desde un punto de vista biológico, los lenguajes naturales, aunque no-optimizados, parecen regirse por una lógica intrínseca a nuestro cerebro que determina su estructura y composición (si esa lógica es universal o determinada por nuestro sistema nervioso es un debate para otra ocasión). La idea no es nueva, Chomsky escribe sobre ella en Teoría de la Gramática Universal [2], donde la enuncia como "el lenguaje humano es el producto de descifrar un programa determinado por nuestros genes". Chomsky lo aplica a la gramática, dejando a la semántica como un problema aparte [3], pero en este trabajo se va a dar un paso más, argumentando que el propio significado del lenguaje responde a estructuras inherentes a nuestra biología.

Esta postura sobre el lenguaje no es única, por supuesto. En directa contraposición tendríamos las teorías ambientalistas, que sugieren que gramática y semántica no son innatas, sino que son una forma vaga de expresar correlaciones entre palabras que escuchamos a nuestro alrededor cuando somos niños. Es en el marco lingüístico de esta teoría donde se vuelven verdaderamente interesantes los recientes descubrimientos en arquitecturas e implementación de redes neuronales, que son maestras en correlacciones. *Generative Pre-trained Transformer*-3 (GPT-3), la última inteligencia artificial desarrollada por OpenAI, está entrenada con cerca de 45TB de información en forma de texto, prácticamente la web publica al completo. Basada en una arquitectura de *transformer*, GPT-3 busca y aprende correlaciones entre todos

estos datos, llegando a simular la sintaxis del lenguaje y con unos resultados sorprendentes a la hora de simular su sentido semántico. Por supuesto, no es perfecta y, aún con la inmensidad en el entrenamiento proporcionado, carece de sentido común a la hora de producir lenguaje.

Aun así, GPT-3 ha generado un entusiasmo tremendo dentro de la comunidad de NLP, pero hay quien defiende que esta falta de sentido común en su funcionamiento está precisamente fundamentada en que su construcción adopta una filosofía ambientalista, olvidando la estructura innata del lenguaje [4]. De hecho, para sus detractores, GPT-3 es un gran experimento que demuestra que las teorías ambientalistas sobre el lenguaje se equivocan (funcionan hasta cierto punto, pero no son lo suficientemente completas).

Es curioso como, desde dos ramas enseñadas de maneras tan distintas, se puede llegar a explorar los entresijos del razonamiento humano de forma tan similar. Este debate sobre las teorías lingüistas se parece mucho al presentado por Judea Pearl [5] en el mundo de la estadística, donde se pretende demostrar que la causalidad no es únicamente un tipo "fuerte" de correlación. Dicho de otra forma, que no todo conocimiento puede ser explicado con relaciones entre datos.

Attention is not all you need es un trabajo que pretende argumentar que el lenguaje de la causalidad es quizás el programa genético inherente al que aludía Chomsky, pero con el matiz de referirse al reino de la semántica.

En cualquier caso, son tiempos apasionantes para el mundo del NLP, la neurociencia, las matemáticas, y el pensamiento humano en general. Cuestiones que han sido planteadas hace siglos tienen una nueva perspectiva de estudio con las construcciones de Aprendizaje Automático Profundo. El desarrollo de los ordenadores ha dotado a los filósofos de un campo de pruebas donde probar sus teorías. En este trabajo se va a explorar este mundo, su construcción matemática y sus implicaciones filosóficas, así como alguna de sus aplicaciones.

El trabajo está estructurado en cuatro capítulos principales, además de la presente introducción y las conclusiones finales.

En el **Capítulo 2** se presenta el campo de la Inteligencia Artificial que simula, aunque sea vagamente, el mecanismo de aprendizaje del cerebro humano: el **Aprendizaje Automático** o *Machine Learning*. Se describe en qué consiste dicho mecanismo y se hace especial hincapié en su capacidad de encontrar correlaciones.

En el **Capítulo 3** se introduce otro campo de la Inteligencia Artificial conocido como **Procesamiento del Lenguaje Natural** o *Natural Language Procesing*, que intenta describir el funcionamiento de los lenguajes naturales con el objetivo de su simulación y comprensión con el uso de ordenadores. En particular, se desarrollan los algoritmos detrás de los *transformers*.

En el **Capítulo 4** se presenta un **ejemplo práctico** de utilización de estas herramientas en el sector de *retail*. Se muestran los avances conseguidos con los *transformers* en una tarea de búsqueda y recuperación de información, pero también se muestran sus limitaciones.

En el **Capítulo 5** se sugiere una posible línea de investigación en la cual se argumenta que las herramientas proporcionadas por la **inferencia causal** puedan ser la pieza (o por lo menos una de ellas) para conseguir "ordenadores que hablen".

Por último, en las **Conclusiones** se hace un resumen de los resultados obtenidos en el ejemplo práctico, así como una reflexión sobre el actual estado del estado del arte en Procesamiento del Lenguaje Natural, Aprendizaje Automático, y las principales vías futuras que se presentan, y que tanto por dificultad como por longitud, no se han podido abordar en este Trabajo de Fin de Grado.

Capítulo 2

Las redes neuronales en el Aprendizaje Automático

El primer paso para entender por completo este trabajo es situarnos en la amalgama de términos con los que somos bombardeados constantemente por parte de los medios. Inteligencia Artificial (AI), Aprendizaje Automático o *Machine Learning* (ML), Aprendizaje Automático Profundo o *Deep Learning* (DL) y redes neuronales o*Neural Networks* (NN) son quizás los que más aparezcan, y para el lector no versado en el tema pudieran parecer sinónimos. Sin embargo, la jerarquía correcta es la que engloba al ML como un campo de estudio dentro de la AI y a las NN como un conjunto de algoritmos dentro del ML. Por su parte, el DL considera solo redes neuronales de alta complejidad computacional.

$$DL \subset NN \subset ML \subset AI$$

2.1 Aprendizaje Automático

La idea fundamental detrás del Aprendizaje Automático es que, teniendo un problema y una hipótesis h para resolverlo, se puede ajustar dicha hipótesis acompañándola de unos parámetros (a veces llamados pesos), y optimizando dichos parámetros utilizando alguna estrategia estadística [6]. Por ejemplo, se puede plantear una función de error que dependa de dichos parámetros y minimizarla, aunque quizás la estrategia más utilizada sea calcular el Estimador de Máxima Verosimilitud, *Maximun Likelihood Estimator* (MLE) en inglés. Sea cual sea la estrategia estadística elegida, se distinguen dos casos principalmente: en el primero se cuenta con el resultado correcto de forma previa, de forma que en cada iteración el resultado de la hipótesis se puede comparar al resultado real; y en el segundo, este resultado correcto es desconocido. En el primer caso se habla de aprendizaje supervisado, y lo que se busca es una función para hacer un mapeo entre el espacio de *inputs* $x^{(i)}$, también llamados características, y el de *outputs* conocidos o etiquetas $y^{(i)}$. En el segundo se habla de aprendizaje no-supervisado, y lo que se busca es aprender la estructura inherente a los datos $x^{(i)}$.

En la metodología de cualquier algoritmo de Aprendizaje Automático se deben ajustar dos elementos: los parámetros de la hipótesis; y la hipótesis en sí, es decir, número de parámetros y naturaleza de los mismos. A este segundo conjunto de valores por ajustar se le denomina conjunto de hiperparámetros. Dependiendo de la función que se utilice como hipótesis y del algoritmo para maximizar su verosimilitud, los hiperparámetros a optimizar podrán ser distintos. Tradicionalmente esta segunda optimización se ha hecho a mano, aunque recientemente hay sistemas que aplican distintos tipos de algoritmos (algoritmos

genéticos, bayesianos, etc) o incluso una red neuronal adicional para esta optimización (AutoML) [1].

En cualquier caso, puesto que solo se dispone de un conjunto de datos, es necesario dividirlo, por lo menos, en dos partes: una para optimizar los parámetros de la función de error/verosimilitud, y otra para optimizar los hiperparámetros de la propia hipótesis. Pero entonces no habría forma de comprobar el funcionamiento del modelo con datos nuevos, por lo que el conjunto de datos inicial se suele dividir en tres partes, la última con el propósito de comprobar cómo generaliza la función de los parámetros e hiperparámetros escogidos a nuevos datos. Esta división se puede realizar de varias maneras, pero los conjuntos se suelen denominar de la misma forma: *entrenamiento*, *validación cruzada* y *test*.

Nótese que en ningún momento se demuestra la veracidad de la hipótesis conjeturada: simplemente se comprueba que tenga un cierto grado de generalización aplicándosela a valores particulares.

2.1.1. Ejemplo Aprendizaje Automático supervisado: regresión lineal

Consideremos el ejemplo concreto de una regresión lineal como mejor ajuste a unos datos.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x, \tag{2.1}$$

donde $h_\theta(x)$ es la función hipótesis, θ^T el vector con componentes los parámetros θ_i traspuesto, y los x_i variables que representan las distintas características a correlacionar. Por simplicidad, se suele escoger $x_0 = 1$.

En este punto quizás al lector le resulte familiar la idea de un ajuste por mínimos cuadrados. Pero, antes de escribir esa función de error y el correspondiente formalismo para optimizarla, es importante notar que los valores de los parámetros θ que optimizan el ajuste con la ec. (2.1) admiten las dos interpretaciones probabilísticas predominantes en la actualidad: la frecuentista y la bayesiana.

Por un lado, pueden considerarse como valores desconocidos, pero fijados por el universo e indiferentes a nuestra opinión. En ese caso, la elección de función de error que se realice para la obtención de dichos valores no los determinará, solo permitirá acercarse a ellos con mayor o menor precisión. La idea es utilizar estrategias estadísticas que permitan el mejor acercamiento. En general, como se ha mencionado, la más utilizada es la del MLE. Esta visión de los parámetros es la visión frecuentista. Recalco: en esta interpretación, la elección de una función de error u otra no necesita justificarse, pues no influye sobre el valor de los parámetros óptimos. Se puede elegir la función habitual de los mínimos cuadrados, u otra que se piense que va a aproximarlos mejor. No obstante, tiene sentido empezar por la función resultante de seguir la estrategia del MLE, pues asegura una buena optimización de base.

En la segunda interpretación no existen unos parámetros fijados por el universo óptimos, sino que están inevitablemente condicionados por la elección que se realice al elegir una estrategia para su obtención. En esta forma de pensar sí que es fundamentalmente importante el definir bien una estrategia estadística, pues dos estrategias distintas no nos acercan más o menos a unos valores óptimos, sino que nos dan valores totalmente distintos. Esta interpretación aparece desarrollada en el Apéndice B. Por extensión del trabajo aquí solo se desarrollará el algoritmo desde un punto de vista frecuentista.

[1] Las herramientas de AutoML son una de la principales ramas en investigación de Aprendizaje Automático [7]. Dos de las librerías más populares que lo implementan, además de los servicios en *cloud*, son `PyCaret` [8] (algoritmos bayesianos) y `TPOT` (algoritmos genéticos) [9].

Regresión lineal con interpretación frecuentista

En esta interpretación se puede plantear cualquier función de error, también llamada función de coste, que se crea adecuada. Se puede elegir, por tanto, la función de coste que sustentaría el tradicional algoritmo de regresión por mínimos cuadradros, ec. (2.2):

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2, \tag{2.2}$$

Definida la función de error, queda minimizarla utilizando alguno de los algoritmos de optimización de funciones. En particular, se puede elegir un método iterativo como el descenso de gradiente, que empieza en algún valor de θ y se actualiza con la ec. (2.3) (esta actualización se realiza simultáneamente con todos los valores de $j = 0, ..., n$).

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{2.3}$$

En la ec. (2.3), a α se le conoce como tasa de aprendizaje. Calculando la derivada parcial con respecto al término θ se obtiene:

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{\partial}{\partial\theta_j}\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \sum_{i=1}^{m}\frac{1}{2m}\frac{\partial}{\partial\theta_j}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$= \sum_{i=1}^{m}\frac{1}{2m}2(h_\theta(x^{(i)}) - y^{(i)})\frac{\partial}{\partial\theta_j}(h_\theta(x^{(i)}) - y^{(i)}) = \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\frac{\partial}{\partial\theta_j}(\sum_{k=0}^{n}\theta_k x_k^{(i)} - y^{(i)})$$

$$= \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Con lo que finalmente se obtiene la regla de aprendizaje de Widrow-Hoff o de los mínimos cuadrados, ec. (2.4).

$$\theta_j := \theta_j + \alpha\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - (h_\theta(x^{(i)}))x_j^{(i)} \tag{2.4}$$

Nótese que, en la ec. (2.4), en cada iteración se tienen en cuenta todos los datos del conjunto de entrenamiento, con lo que se podrá llegar a tener un coste computacional considerable. A este tipo de descenso de gradiente, donde se consideran todos los datos del entrenamiento, se le conoce como descenso de gradiente en lote o *batch gradient descent*. Una alternativa es considerar la suma de solo $k \in \mathbb{N}$ de sus términos, como se muestra la ec. (2.5), lo que se conoce como descenso de gradiente en mini-lotes o *mini-batch gradient descent*. Dentro de esta, se distingue cuando $k = 1$ y se le denomina descenso de gradiente estocástico.

$$\theta_j := \theta_j + \alpha\frac{1}{k}\sum_{i=1}^{k}(y^{(i)} - (h_\theta(x^{(i)}))x_j^{(i)}, \quad k < m \tag{2.5}$$

Por supuesto, con la ec. (2.5) la dirección de descenso no estará tan optimizada como con la ec. (2.4), y puede que no llegue a converger nunca al mínimo, pero en algunos casos pueda merecer la pena si el número de datos de entrenamiento es muy alto. Al final, no hacer la suma supone una reducción en el

coste computacional y, además, en la práctica los valores a los que converge, cercanos al mínimo, son igualmente válidos.

Se ha mencionado antes que, aunque con una interpretación frecuentista de los parámetros θ no es necesario justificar la elección de una función de coste u otra, es cierto que una manera razonable de hacer la elección es derivándola desde la estrategia estadística del MLE. En esta primera aproximación se ha introducido una función de coste *ad hoc*, ec. (2.2). Ahora, se van a hacer las estimaciones probabilísticas necesarias para poder seguir la metodología del MLE.

La primera asunción que se puede hacer es que puede haber algo de ruido aleatorio, o características desconocidas, en el mapeo que se realiza entre los *inputs* y *outputs*. Para tenerlo en cuenta se puede introducir un error $\epsilon^{(i)}$ en la expresión de regresión lineal.

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

Se puede asumir también que los $\epsilon^{(i)}$ son independientes e idénticamente distribuidos de acuerdo a una distribución Gaussiana (también llamada distribución normal) con media 0 y varianza σ^2.

$$\epsilon^{(i)} \sim N(0, \sigma^2)$$

Su densidad de probabilidad estará dada entonces por:

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right),$$

lo cual a su vez implica,

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

donde $p(y^{(i)}|x^{(i)};\theta)$ es la distribución de $y^{(i)}$ dados los $x^{(i)}$ y parametrizada por θ. Se puede expresar también como $p(y^{(i)}|x^{(i)};\theta) \sim N(\theta^T x^{(i)}, \sigma^2)$. Recuérdese que en esta visión frecuentista θ no es una variable aleatoria, así que no se puede condicionar la probabilidad de los *outputs* con ella, es decir, no se puede escribir $p(y^{(i)}|x^{(i)}, \theta)$.

Sea X la matriz que por filas contiene cada $x^{(i)}$ e $\vec{y}$ el vector que contiene cada $y^{(i)}$. La probabilidad de los datos está dada entonces por $p(\vec{y}|X;\theta)$. Esta cantidad se ve normalmente como una función de $\vec{y}$ para un valor de θ fijo. Cuando se quiere ver explícitamente como una función de θ se le llama verosimilitud L (del inglés *likelihood*):

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y}|X;\theta)$$

Y con la asunción de unos $\epsilon^{(i)}$ independientes se puede escribir la ec. (2.6):

$$L(\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \tag{2.6}$$

Teniendo este modelo probabilístico, una forma razonable para elegir los parámetros θ es elegirlos de forma que los datos sean lo más verosímiles posible. Esto mismo es lo que postula el principio de Máxima Verosimilitud, y la forma de llevarlo a cabo es maximizando $L(\theta)$. Generalmente, puesto que se puede maximizar $L(\theta)$ maximizando cualquier función estrictamente creciente de ella, se elige su logaritmo $l(\theta)$ para hacer el cálculo.

$$l(\theta) = \log L(\theta)$$

$$= \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$
$$= \sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$
$$= m\left(\log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} * \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2\right)$$

Por tanto, maximizar $l(\theta)$ es equivalente a minimizar

$$\frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2$$

que coincide con la ec. (2.2).

En resumen, aunque no es necesario justificar la elección de coste elegida en esta interpretación frecuentista, sí que es razonable escoger una que maximice la verosimilitud, y con estas asunciones probabilísticas se deriva de forma natural la función de coste de los mínimos cuadrados.

2.1.2. *Overfitting*, *underfitting* e hiperparámetros.

Hasta este momento se ha introducido un ejemplo de la metodología para ajustar datos con una interpretación frecuentista de los parámetros. De acuerdo al método descrito, dado un conjunto de entrenamiento, se pueden optimizar los parámetros de una función hipótesis utilizando la estrategia estadística de maximizar la verosimilitud. Pero, ¿no es el mejor ajuste de unos datos el que nos da la función que pasa por cada uno de ellos? Sí, pero no solo se trata de ajustar los datos con los que entrena el modelo, sino de realizar predicciones con ellos. Para este fin, "mejor" es una palabra con trampa en estadística. Como mínimo, se debe distinguir entre precisión y exactitud para poder distinguir la calidad de las predicciones de los distintos modelos.

A menudo se utiliza el caso de una regresión polinomial (ec. (2.7)) como ejemplo para ilustrar el *overfitting* y el *underfitting*. En este tipo de regresión, en vez de suponer una función hipótesis que dependa linealmente de varias variables independientes que representen distintas características, se supone una función hipótesis que dependa polinomialmente (no necesariamente lineal) de una sola variable.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x^i \tag{2.7}$$

Un polinomio de mayor grado ajustará con mejor precisión los datos de entrenamiento, pero si se toman nuevos datos, podría generalizar con poca exactitud. Se tendría entonces *overfitting*. Por otro lado, es posible también que en la búsqueda de un polinomio con un grado lo suficientemente pequeño para permitir una generalización para datos nuevos, se obtenga un ajuste poco preciso, *underfitting*. Los dos fenómenos vienen representados en la Figura 2.1, en la fila correspondiente "Regression: ilustration".

¿Cómo se busca entonces el grado n del polinomio? Podemos pensar en él como un parámetro más que se debe ajustar. Este tipo de parámetros que determinan nuestra función hipótesis, y cuyo nivel de abstracción es distinto al de los parámetros que la constituyen, son los mencionados hiperparámetros. Es natural en este punto preguntarse, ¿pueden estos hiperparámetros optimizarse como se optimizaban los parámetros? No exactamente, no son parámetros de ninguna función conocida, por tanto, inferir

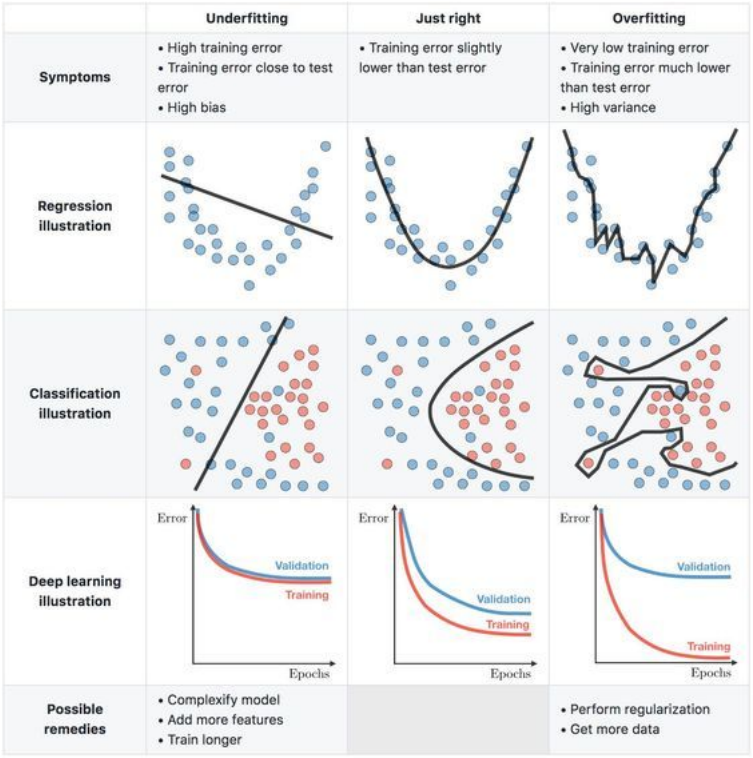

Figura 2.1: *Overfitting* vs *underfitting* [6]

una función de coste derivando como se hacía con los θ no es posible. Lo que se tiene es un problema de "Optimización de caja negra", *Black-box Optimization* (BBO). Existen distintos tipos de algoritmos BBO, algunos de los cuales se utilizan en este problema, como los ya mencionados algoritmos genéticos y algoritmos los de optimización bayesiana, además de algunos sistemas de redes neuronales recurrentes. Sin embargo, lo más común es optimizar estos hiperparámetros "a mano", literalmente con una metodología de prueba y error donde se estudia, utilizando unas herramientas de las que se va a hablar a continuación, que no se produzcan los fenómenos ni de *overfitting* ni de *underfitting*.

2.1.3. Sesgo y varianza.

Las herramientas para determinar si el modelo tiene *overfitting* o *underfitting* son el sesgo o *bias* y la varianza. Entre estas dos variables estadísticas se produce un intercambio en el que se intenta alcanzar un equilibrio. La mejor forma de visualizar el significado de estas variables, es representando los parámetros θ_i de un θ bidimensional, el uno frente al otro, tal y como aparece en la Figura 2.2.

Una varianza alta posibilitará mayor precisión en el ajuste de los datos, pero a cambio, perderá exactitud al generalizar. Significará por tanto que el modelo está teniendo *overfitting*. Por su parte, un alto sesgo concentrará los valores de los parámetros, ganando en exactitud al generalizar, pero con la posibilidad de hacerlo con una precisión pobre, teniendo por tanto *underfitting*.

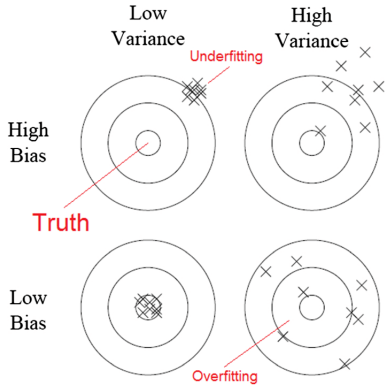

Figura 2.2: Intercambio sesgo/varianza [6]

2.2 Redes neuronales

Se ha abordado el ejemplo de la regresión lineal para ilustrar el *modus operandi* con cualquier algoritmo de Aprendizaje Automático. En particular, se pueden considerar a las redes neuronales como un subconjunto de los mismos. Pero, ¿qué es exactamente una red neuronal? Cuando empecé a estudiar este tema, a menudo escuchaba conversaciones en las que las redes neuronales se presentaban como esta caja negra capaz de aprender. Después de años programando desde una perspectiva simbólica, esto me era totalmente contra-intuitivo: ¿cómo iba un algoritmo, por bueno que fuese, a escaparse de la lógica implementada por su programador? Esto para mí sonaba mucho a originalidad, creación, inventiva, arte, y un largo etcétera de atributos estrechamente relacionados con la psicología. ¿Tendría el nombre "red neuronal" algo que ver con esto, y con una posible semejanza entre los algoritmos correspondientes y las neuronas de nuestro cerebro? Creo que fue esta visión romántica/filosófica de las redes neuronales la que me atrajo a mí y a mucha otra gente a interesarnos por este campo del Aprendizaje Automático.

2.2.1. ¿Qué es una red neuronal?

Una red neuronal es un modelo computacional que busca obtener correlaciones entre datos. Para ello, se basa en la creación de nodos, dispuestos en capas, y conexiones entre los nodos de (por lo general) distintas capas. Con esto se consigue una concatenación de operaciones sencillas que simulan patrones mucho más complejos. Por lo menos remotamente, estos nodos se asemejan a las neuronas en el cerebro (aunque eso es otro tema de debate), de ahí que se les conozca como neuronas. El Aprendizaje Automático de la red consiste en averiguar con qué intensidad se hablan cada una de las neuronas entre si: cada comunicación entre neuronas esta representada con un "peso". Al final del proceso, se obtiene un modelo de pesos y neuronas al que se le pueden introducir unos datos y que devuelve unos resultados como si de una función se tratase. Cuando antes se mencionaba que se suele hablar de las redes neuronales como cajas negras capaces de aprender, la parte de caja negra se refiere precisamente a esto: al final del proceso se tiene algo que actúa como una función pero no se sabe exactamente de que función se trata (y por tanto, tampoco se puede programar). Nótese que su proceso de aprendizaje sin embargo está perfectamente definido, por lo tanto esa es la parte que se programa. En [10] se habla sobre los Teoremas de Aproximación Universal, y como gracias a ellos se demuestra que, bajo ciertas condiciones, cualquier función puede aproximarse (al menos teóricamente) con una red neuronal. Dicho de otra forma, sea cual sea la relación entre los datos que se quiere obtener, al menos teóricamente se podría realizar con este tipo de modelo computacional.

2.2.2. Definición formal

Una red neuronal artificial, *Neural Network* (NN), se define formalmente (véase [11]) como una tripla

$$N = (D, f_i, A)$$

donde:

- D es un digrafo numerable, localmente finito y con arcos etiquetados. Sus vértices se corresponden con las neuronas, sus arcos a las conexiones sinápticas y las etiquetas de los mismos a los pesos, que representan las intensidades de las conexiones sinápticas entre neuronas. La notación que se va a utilizar es ω_{ij} para referirse al peso del arco desde la neurona j hasta la neurona i.

- A es un conjunto que contiene los elementos de entrada de las unidades conocido como "conjunto de activación". Una entrada típica a una neurona está dada por:

  $$net_i = \sum_j \omega_{ij} x_j$$

  donde x_j es el valor de salida de la unidad j.

- $\{f_i : A \longrightarrow A\}_{i \in V}$ es una colección de funciones conocidas como "funciones de activación" o "funciones de transferencia" (con V conjunto de vértices del grafo D).

Localmente la red tiene la siguiente dinámica:

$$x_i := f_i(net_i - \theta_i)$$

donde θ_i es el *valor de umbral* que indica, en el caso más sencillo, que la neurona se activa si la entrada excede este valor y permanece inactiva en caso contrario.

En toda red neuronal se consideran tres tipos de unidades:

1. *Unidades de entrada*: reciben los valores de entrada del sistema.
2. *Unidades de salida*: contienen los valores finales calculados por la red neuronal después de cada proceso computacional.
3. *Unidades ocultas*: no tienen comunicación con el exterior de la red neuronal, el sistema las utiliza para representaciones internas.

Aunque de forma general se definen entradas a neuronas arbitrarias, la entrada más utilizada es la mencionada $net_i = \sum_j \omega_{ij} x_j$, es decir la entrada lineal. Para que la red pueda aproximar funciones no lineales se eligen funciones de activación con un patrón definido: cambios suaves en su comportamiento en la mayoría de su dominio, con un cambio brusco en un intervalo pequeño. El por qué de elegir este tipo de funciones responde a los teoremas de aproximación universal y aparece intuitivamente explicado en [12] [2].

[2] En esta tesitura, se puede interpretar el comportamiento de cada capa de la red como una nueva representación de los datos [13], obtenida a partir de la representación realizada por la capa anterior como: una transformación lineal realizada por los pesos, una traslación realizada por los valores umbrales, y una transformación no lineal del espacio resultante mediante la función de activación.

Funciones de activación para redes de entrada lineal

Aunque hay muchos tipos de funciones de activación, algunas de las más comunes son:

- **Función sigmoide o logística:** Función que toma valores entre 0 y 1 definida por la ec. (2.8). Es lenta en el aprendizaje, por lo que ya no se usa en redes neuronales profundas.

$$f(x) = \frac{1}{1 - e^{-x}} \tag{2.8}$$

- **Función tangencial hiperbólica**: Toma valores entre -1 y 1 y responde a la ec. (2.9).

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.9}$$

- **ReLU**: Las siglas se corresponden con *Rectified Linear Unit*. Es la función más utilizada en la actualidad, responde a la ec. (2.10).

$$ReLU(x) = \max(0, x) \tag{2.10}$$

- **Softmax**: Función de activación principalmente utilizada para clasificaciones. Es una generalización de la función logística que viene expresada en la ec. (2.11).

$$softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.11}$$

  donde $\boldsymbol{z} = (z_1, ..., z_K) \in \mathbb{R}^K$ y $K \in \mathbb{N}$ suele ser el número de clases a clasificar.

Estas funciones son las que se han utilizado de forma clásica, pero existen variaciones como *GeLU* y *sneaky ReLU* entre muchas otras con las que se trabaja actualmente.

2.2.3. Arquitecturas de redes neuronales

En función de la tarea que se quiera realizar, existen distintas formas de relacionar las neuronas entre sí. Entre las más conocidas se encuentran:

Perceptrón multicapa (MLP)

La red neuronal más básica se conoce como perceptrón multicapa (MLP). En este tipo de red neuronal las neuronas se organizan en capas, y las unidades de una capa se conectan con unidades de las capas siguientes, siendo el patrón más utilizado el que solo considera conexiones con la capa inmediatamente posterior. Si todas las neuronas de una de las capas tienen conexiones con todas las neuronas de la capa anterior, se le suele denominar "capa densa".

Redes neuronales recurentes (RNN)

Utilizadas para el estudio de datos secuenciales, se basan en utilizar en el cómputo de cada capa un término que proporcione directamente información sobre las capas anteriores. En el Apéndice C se proporciona una explicación detallada de las mismas.

Redes convolucionales (CNN)

Redes utilizadas de forma principal en el procesamiento de imágenes, aunque como se verá en el Capítulo 5, no restringidas al mismo. Su desarrollo teórico formal excede a la longitud de este trabajo.

Capítulo 3

Transformers en el NLP

El *Procesamiento del Lenguaje Natural* (NLP) es un campo de la Inteligencia Artificial que trata de posibilitar la comunicación entre humanos y máquinas a través de los lenguajes naturales. El planteamiento tradicional a este problema, inspirado por teorías lingüísticas como la de Chomsky, ha consistido en definir una serie de reglas que permitiesen a los ordenadores construir frases, oraciones y, en general, texto. La realidad es que la enorme cantidad de excepciones gramaticales, palabras polisémicas y homónimas, conjugaciones, etcétera, ya hacen de este planteamiento algo tremendamente difícil de resolver. Si a ello le sumamos que no sólo se trata de crear texto sintácticamente correcto, sino de crear texto con sentido semántico, el problema alcanza unas proporciones inconmensurables. Por ejemplo, "este ordenador es una manzana por la tarde" es una oración sintácticamente correcta, pero, ¿semánticamente?. El contexto tendría que ser por lo menos extraño para que lo fuera. En realidad, el ejemplo que se suele ofrecer de oración sintácticamente correcta sin un significado discernible se debe también a Chomsky y es "*Colorless green ideas sleep furiously.*" [3].

Es por eso que desde la explosión del *Deep Learning* con la llegada de las GPUs (*Graphical Process Units*) se ha buscado la forma de aplicar redes neuronales a este problema. La idea de base es simple: alimentar modelos con todo el texto escrito posible e intentar extraer las reglas sintácticas y el sentido semántico del lenguaje como correlaciones entre los datos utilizados. Este *modus operandi* viene a representar lo que los lingüistas conocen como "teorías ambientalistas".

En un primer momento había tareas diferenciadas sobre las que se intentaban aplicar estos conceptos: análisis de sentimiento, búsqueda y recuperación de información, traducción, etc. Sobre estas se dirigía el entrenamiento de redes neuronales, por lo general recurrentes, de una forma supervisada (se etiquetaban conjuntos de datos a mano). Con la llegada de la arquitectura de los *transformers* y su fácil paralelización se pasó a una dinámica auto-supervisada en la que las tareas para las que se entrenaban estas redes empezaron a ser más generalistas. Así, por ejemplo, a la arquitectura desarrollada por Google se la entrenaba para auto-completar frases a las que se les quitaban palabras aleatoriamente y a la de OpenAI se le asignaba la tarea de predecir el final de oraciones [14]. Precisamente, son estas dos tareas las que definen las dos grandes familias de modelos del lenguaje basados en la arquitectura *transformers* de la actualidad: la familia de los *bidirectional encoders* (BERT [1], RoBERTa [15]) y la familia de los modelos *decoders* autorregresivos (GPT , Gopher)[14]. Ya no era necesario un etiquetado a mano de conjuntos de datos, sino que simplemente bastaba con proveer textos para que los modelos aprendiesen. Es así como nacieron los "grandes modelos del lenguaje", alimentados con prácticamente toda la web pública.

Los resultados son sorprendentes: a pesar de haber sido entrenados en tareas tan sencillas, resulta que su desempeño en las tareas tradicionales superaban a los modelos específicamente entrenados para ellas.

Sin embargo, estamos aún lejos de "hablar" con los ordenadores, y quizás nunca lleguemos a hacerlo siguiendo esta metodología.

El problema es profundo, y posiblemente no esté únicamente recluido a los dominios del Procesamiento de Lenguaje Natural. De alguna forma el conocimiento *a priori* y a *posteriori* del que hablaba Kant en *Crítica a la razón pura* [16] está relacionado con él. Pero es en el debate de la causalidad donde yo personalmente encuentro las mayores similitudes y, quizás, la mejor baza sobre qué pieza le falta a los modelos del lenguaje actuales.

El mundo del Procesamiento de Lenguaje Natural es demasiado amplio como para abarcarlo en un Trabajo de Fin de Grado. Este capítulo se centra principalmente en explicar la estructura de los *transformers* actuales.

3.1 *Transformers*

Desde hace años el enfoque más habitual con Aprendizaje Automático al problema del procesamiento de lenguaje natural es aplicar modelos secuenciales. En el apéndice C se explica cómo hacerlo con redes neuronales recurrentes (RNN), la primera arquitectura utilizada en este tipo de problemas. También se menciona como estas, así como sus versiones con *gates*, sufren del problema de *corta memoria* en menor o mayor medida. Además, presentan también el problema de que son difícilmente paralelizables.

Fue en 2017, en un *paper* de título *Attention is all you need* cuando se introdujo por primera vez la arquitectura de red neuronal que acabaría dominando por completo el panorama del NLP en los últimos años: los *transformers*. Estos siguen sobresaliendo en el procesamiento de datos secuenciales a la par que consiguen solventar los dos problemas de las RNN con la arquitectura presentada en la Figura 3.1.

En el nivel más superficial, un *transformer* está formado por un *enconder*, que representa un *input*, como una representación de valores continuos en un espacio vectorial que contenga toda la información aprendida sobre ese *input*. Luego, esta representación continua es introducida en un *decoder* que, utilizando también el *output* previo, genera un solo *output* para el *input* introducido.

Antes se ha mencionado que existen dos familias principales de modelos basados en la arquitectura *transformer*: los *bidirectional encoders* y los *autoregressive decoders*. Estas familias de modelos abordan dos tareas simples distintas, y lo hacen de manera que modifican la arquitectura *transformer* de la forma más conveniente en cada caso. Como su nombre indica, los *bidirectional encoders* adoptan la estructura del *encoder* y hacen superposiciones con ella excluyendo la parte del *decoder*. Por otra parte, los *autoregresive decoders* hacen el análogo con el *decoder*. A la primera familia se la conoce también por el nombre de *masked language models* y a la segunda como *language models*. Por su parte la estructura original *encoder-decoder* representada en la Figura 3.1 está implementada en BART (*Bidirectional Autoencoders Representation from Transformers*) y es particularmente útil para la tarea de traducción.

Tokenización

El primer paso para cualquier tipo de modelo en NLP es la tokenización, esto es, la selección de las piezas de texto que se van a introducir como *inputs* a los distintos modelos. Existen tres tipos de tokenización principales: la tokenización por caracteres, la tokenización por palabras y la tokenización por subpalabras, siendo la última la más comúnmente utilizada en los últimos años.

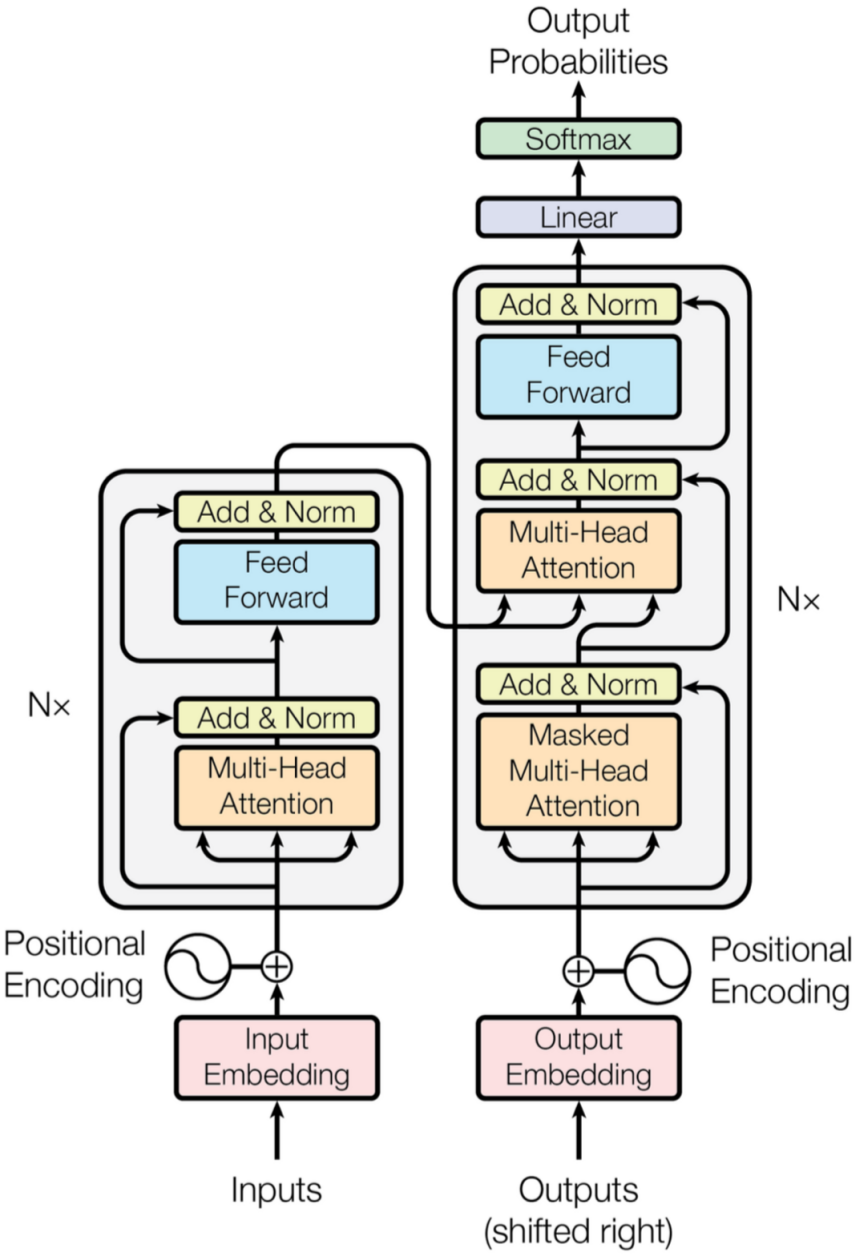

Figure 1: The Transformer - model architecture.

Figura 3.1: Arquitectura de una red neuronal *transformer* presentada en *Attention is all you need*[1]

3.1.1. Notación

A lo largo de este trabajo, para cualquier conjunto S de cardinal $|S| = N_S$, se denota $[N_S] = \{1, ..., N_S\}$.

Sea V un conjunto finito al que llamaremos *vocabulario*, con elementos $\{v_i\}_{i\in[N_V]}$. Los elementos de este conjunto pueden ser los distintos tókenes mencionados con anterioridad: caracteres, palabras o más comúnmente sub-palabras. Sea $x = x_1x_2...x_l \in V^*$, donde $x_1, x_2, ..., x_l \in V$ y V^* es la clausura de V, una

secuencia de $l \in \mathbb{N}$ tókenes (una oración, un párrafo o un documento por ejemplo).

En una matriz $W \in \mathbb{R}^{n \times m}$ para referir todos los elementos de una fila se denota $W[i,:]$, $i \in \{1, ..., m\}$ y para referir todos los elementos de una columna $W[:,j]$, $j \in \{1, ..., n\}$.

3.1.2. Tareas típicas (del NLP)

Procesamiento secuencial *(sequence modeling)*

Dado un vocabulario V y un conjunto secuencias de tókenes que se suponen muestreadas de forma independiente e idénticamente distribuidas por alguna función de distribución P sobre V^*, el objetivo es aprender un estimador $\hat{P}$ (por ejemplo el MLE) de la distribución $P(x)$. En la práctica, $\hat{P}$ se suele descomponer según la regla de la cadena en $\hat{P}(x) = \hat{P}(x_1, ..., x_l) = \hat{P}_\theta(x_1) \cdot \hat{P}_\theta(x_2|x_1) \cdots \hat{P}_\theta(x_l|x_1, ..., x_{l-1})$, donde θ son los parámetros de la red neuronal. Es decir, el objetivo es aprender la distribución sobre un único token x_t dados como contexto todos los tókenes $x_1, ..., x_{t-1}$ que le preceden. Este tipo de tarea engloba al modelado de lenguaje y la generación de música entre otras.

Predicción *sequence-to-sequence* (seq2seq)

Dado un vocabulario V y un conjunto de parejas de datos secuenciales, independientes e idénticamente distribuidos $(z^{(n)}, x^{(n)}) \sim P$, donde P es una distribución sobre $V^* \times V^*$, el objetivo es aprender un estimador $\hat{P}$ sobre la distribución condicional $P(x|z)$. En la práctica, $\hat{P}$ se suele descomponer según la regla de la cadena en $\hat{P}(x|z) = \hat{P}(x_1, ..., x_l|z) = \hat{P}_\theta(x_1|z) \cdot \hat{P}_\theta(x_2|x_1, z) \cdots \hat{P}_\theta(x_t|x_1, ..., x_{t-1}, z)$. Como ejemplos, se incluyen: traducción (z = una oración en inglés, x = la misma frase en español), *question answering* (z = la pregunta, x = la respuesta correspondiente) y *text-to-speech* (z = un texto, x = una voz grabada recitando el texto).

Clasificación

Dado un vocabulario V, un conjunto de clases C y un conjunto de pares (secuencia, clase) independientes e idénticamente distribuidos muestreados desde la distribución P, el objetivo de la clasificación es obtener un estimador de la distribución condicional $P(c|x)$, donde $c \in C$ y $x \in V^*$. La clasificación de sentimientos y el filtrado de *spam* son ejemplos de este tipo de tareas.

3.1.3. Arquitecturas: componentes

Para cada una de las tareas anteriores existen distintas arquitecturas, pero los bloques (funciones con parámetros de aprendizaje) de los que se componen son los mismos. [1]

Token *embedding*

Este bloque es el encargado de aprender cómo representar cada token en el vocabulario como un vector en $\mathbb{R}^{d_e}$, donde d_e es la dimensión de la representación vectorial (el *embedding*) de un token. Tiene como

[1] Vale la pena mencionar que la bibliografía al respecto de los algoritmos detrás de cada bloque ha sido bastante críptica desde la publicación de *Attention is all you need*, y sólo es recientemente qué se ha publicado una versión formal de los mismos [14].

input el identificador i de un token $v_i \in V$ y como *output*, la i-ésima columna, $\boldsymbol{W_e}[:, i]$, de la matriz *embedding* de los tókenes $\boldsymbol{W_e} \in \mathbb{R}^{d_e \times N_V}$. Abusando de la notación, en el resto del documento se utilizará $\boldsymbol{W_e}[:, v_i]$ para referirse a la columna correspondiente al índice que representa la posición del token v_i en V.

Positional embedding

En el artículo original de *Attention is all you need* se elegía como *positional embedding* una matriz $\boldsymbol{W}_p$ con d_e filas y l_{max} columnas definida por la ec. (3.1).

$$\begin{aligned} \boldsymbol{W}_p[2i-1, t] &= \sin\left(\frac{t}{l_{max}^{2i/d_e}}\right) \\ \boldsymbol{W}_p[2i, t] &= \cos\left(\frac{t}{l_{max}^{2i/d_e}}\right) \end{aligned} \tag{3.1}$$

para $0 < i \leq d_e/2$ y $0 < t \leq l_{max}$.

La intuición detrás de estas fórmulas es que, por cada posición par, se asigna un vector calculado con la función coseno y, por cada posición impar uno con la función seno. Se eligen estas funciones en tándem porque tienen propiedades lineales que facilitan al modelo el aprendizaje de atención. Teóricamente un mapeo no aprendido como este permite manejar secuencias arbitrariamente largas.

Los *transformers* más modernos cuentan con un bloque con sus propios parámetros de aprendizaje. Es el bloque encargado de aprender cómo representar la posición de un token en una secuencia como un vector de $\mathbb{R}^{d_e}$. Por ejemplo, la posición del primer token se representa con un vector (aprendido) $\boldsymbol{W}_p[:, 1]$, la posición del segundo token se representa con otro vector (aprendido) $\boldsymbol{W}_p[:, 2]$ y, en general, la posición del l-ésimo token se representa con un vector $\boldsymbol{e}_p = \boldsymbol{W}_p[:, l]$. Los bloques de *positional embeddings* aprendidos requieren que la secuencia introducida como *input* tenga una longitud máxima l_{max} para tener un tamaño de matriz finito y prefijado antes del entrenamiento.

El propósito del *positional embedding* es permitir al *transformer* tener en cuenta la naturaleza secuencial de los tókenes en la oración; sin él, la representación sería invariante ante permutaciones y simplemente tendríamos un modelo que interpretaría las frases como "bolsas de palabras".

El *positional embedding* de un token se suele sumar al *token embedding* para formar el que se conoce como *input embedding*, ec. (3.2).

$$\boldsymbol{e} = \boldsymbol{W}_e[:, x_t] + \boldsymbol{W}_p[:, t] \tag{3.2}$$

Attention

El bloque de *atención* es el bloque principal de cualquier *transformer*. Representado en el Algoritmo 1, permite a la red neuronal aprender el significado semántico de una oración en su conjunto para predecir el token correspondiente.

En su nivel más superficial, la "atención" se basa en distinguir entre el token a representar $x_o \in V$ y los tókenes de contexto $T = \{z_t \in V\}_{t \in [N_T]}$, y en qué grado se relaciona el uno con los otros. Para ello, aprende un mapeo desde el *input embedding* de x_o hasta un vector $\boldsymbol{q} \in \mathbb{R}^{d_{attn}}$ denominado vector *query*. Se aprenden otros dos mapeos desde el *input embedding* de cada z_t, hasta dos vectores $\boldsymbol{k}_t \in \mathbb{R}^{d_{attn}}$ y $\boldsymbol{v}_t \in \mathbb{R}^{d_{value}}$, denominados *key* y *value* respectivamente. Se interpreta que la importancia del token z_t sobre el token x_o viene reflejada en los productos internos $\boldsymbol{q}^T\boldsymbol{k}_t$, denominados *scores* (se utilizan para derivar una distribución sobre los tókenes contextuales, que después a su vez se combina con los vectores *value*).

Algoritmo 1: Atención básica para un único token.

Input: $\boldsymbol{e} \in \mathbb{R}^{d_{in}}$, *input embedding* del token a representar x_o.
Input: $\boldsymbol{e}_t \in \mathbb{R}^{d_{in}}$, *input embedding* del token de contexto $z_t \in T$.
Output: $\tilde{\boldsymbol{v}} \in \mathbb{R}^{d_{out}}$, la representación vectorial conjunta del token x_o y su contexto T.
Parámetros: $\boldsymbol{W_q}, \boldsymbol{W_k} \in \mathbb{R}^{d_{attn} \times d_{in}}$, $\boldsymbol{b_q}$, $\boldsymbol{b_k} \in \mathbb{R}^{d_{attn}}$; proyecciones lineales asociadas a *query* y a *key*.
Parámetros: $\boldsymbol{W_v} \in \mathbb{R}^{d_{out} \times d_{in}}$, $\boldsymbol{b_v} \in \mathbb{R}^{d_{out}}$; la proyección lineal asociada a *value*
1 $\boldsymbol{q} \longleftarrow \boldsymbol{W_q e} + \boldsymbol{b_q}$
2 $\forall t \in [N_T] : \boldsymbol{k}_t \longleftarrow \boldsymbol{W_k e}_t + \boldsymbol{b_k}$
3 $\forall t \in [N_T] : \boldsymbol{v}_t \longleftarrow \boldsymbol{W_v e}_t + \boldsymbol{b_v}$
4 $\forall t \in [N_T] : \alpha_t = \frac{\exp\left(\boldsymbol{q}^T \boldsymbol{k}_t / \sqrt{d_{attn}}\right)}{\sum_u \exp\left(\boldsymbol{q}^T \boldsymbol{k}_u / \sqrt{d_{attn}}\right)}$
5 return $\tilde{\boldsymbol{v}} = \sum_{t=1}^{T} \alpha_t \boldsymbol{v}_t$

Tabla 3.1

Para poder generalizar el mecanismo de atención de un token a un secuencia completa de tókenes, se distingue entre secuencia primaria $\boldsymbol{x}$, con $l_X \in \mathbb{N}$ tókenes de índices $t_X \in [l_X]$; y secuencia de contexto $\boldsymbol{z}$, con $l_Z \in \mathbb{N}$ tókenes de índices $t_Z \in [l_Z]$. La secuencia primaria incluye todos los tókenes a representar, mientras que la secuencia de contexto incluye todos los tókenes que formarán parte del contexto de cada token de la secuencia primaria (no todos los tókenes de $\boldsymbol{z}$ se utilizan en cada token de $\boldsymbol{x}$, de ahí el definir un conjunto de contexto T para el algoritmo anterior). Los *input embeddings* de los tókenes de cada una de estas secuencias se recogen en dos matrices $\boldsymbol{X} \in \mathbb{R}^{d_X \times l_X}$ y $\boldsymbol{Z} \in \mathbb{R}^{d_Z \times l_Z}$, donde d_X es la dimensión de los *input embeddings* de $\boldsymbol{x}$ y d_Z la dimensión de los *input embeddings* de $\boldsymbol{z}$.

Además, se necesitan introducir dos ecuaciones para terminar de presentar la generalización del mecanismo de atención. Por un lado la ec. (3.3) presenta la función *softmax* para matrices. Por su parte la ec. (3.4) presenta la función máscara, donde la notación $a = [[b]]$ indica $a = 0$ si b es falso y $a = 1$ si b es verdadero.

$$softmax(\boldsymbol{A}[t_Z, t_X]) := \frac{\exp(A[t_Z, t_X])}{\sum_t \exp(A[t, t_X])}, \qquad (\forall \quad t_Z \in [l_Z], t_X \in [l_X]) \tag{3.3}$$

$$Mask[t_Z, t_X] = \begin{cases} 1 \text{ para atención bidireccional} \\ \\ [[t_Z \leq t_X]] \text{ para atención unidireccional} \end{cases} \tag{3.4}$$

Con estas herramientas, el mecanismo de atención generalizado se presenta en el Algoritmo 2 (en el mismo $\boldsymbol{1}^T$ es un vector fila de unos para permitir la suma de matrices). Alterando los valores de $\boldsymbol{X}, \boldsymbol{Z}$ y $Mask$ que se le introducen como *input*, se pueden describir los mecanismos de atención más comunes:

Atención bidireccional (*Bidirectional/Unmasked self-attention*)

Dada una secuencia de tókenes $\boldsymbol{x}$ aplica el mecanismo de atención a cada token utilizando todos los demás tókenes de la secuencia como contexto. Es decir, las secuencias primaria y de contexto coinciden, $\boldsymbol{x} = \boldsymbol{z}$ y, por ende, $\boldsymbol{Z} = \boldsymbol{X}$ (lo que le da el nombre de *self-attention*). Además, $Mask \equiv 1$, de ahí el "bidireccional".

Algoritmo 2: $\tilde{\boldsymbol{V}} \longleftarrow$ Atención$(\boldsymbol{X}, \boldsymbol{Z} | \mathcal{W}_{qkv}, \text{Mask})$

/* Calcula una única *head* de *self-* o *cross-attention* (*con máscara*). */

Input: $\boldsymbol{X} \in \mathbb{R}^{d_X \times l_X}$, $\boldsymbol{Z} \in \mathbb{R}^{d_Z \times l_Z}$ los *input embeddings* de la secuencias primaria y de contexto.

Output: $\tilde{\boldsymbol{V}} \in \mathbb{R}^{d_{out} \times l_X}$ la representación vectorial conjunta de los tókenes en $\boldsymbol{X}$ y su contexto $\boldsymbol{Z}$.

Parámetros: $\mathcal{W}_{qkv}$ formada por: $\boldsymbol{W_q} \in \mathbb{R}^{d_{attn} \times d_X}$, $\boldsymbol{b_q} \in \mathbb{R}^{d_{attn}}$.
$\boldsymbol{W_k} \in \mathbb{R}^{d_{attn} \times d_Z}$, $\boldsymbol{b_k} \in \mathbb{R}^{d_{attn}}$.
$\boldsymbol{W_v} \in \mathbb{R}^{d_{out} \times d_Z}$, $\boldsymbol{b_v} \in \mathbb{R}^{d_{out}}$.

Hiperparámetros: Mask $\in \{0,1\}^{l_Z \times l_X}$

1 $\boldsymbol{Q} \longleftarrow \boldsymbol{W_q X} + \boldsymbol{b_q} \boldsymbol{1}^T$, ▷ $\boldsymbol{Q}uery \in \mathbb{R}^{d_{attn} \times l_X}$

2 $\boldsymbol{K} \longleftarrow \boldsymbol{W_k Z} + \boldsymbol{b_k} \boldsymbol{1}^T$, ▷ $\boldsymbol{K}ey \in \mathbb{R}^{d_{attn} \times l_Z}$

3 $\boldsymbol{V} \longleftarrow \boldsymbol{W_v Z} + \boldsymbol{b_v} \boldsymbol{1}^T$, ▷ $\boldsymbol{V}alue \in \mathbb{R}^{d_{out} \times l_Z}$

4 $\boldsymbol{S} \longleftarrow \boldsymbol{K}^T \boldsymbol{Q}$, ▷ $\boldsymbol{S}core \in \mathbb{R}^{l_Z \times l_X}$

5 $\forall t_Z, t_X$ si Mask$[t_X, t_Z] \neq 1$ entonces $\boldsymbol{S}[t_X, t_Z] \longleftarrow -\infty$

6 return $\tilde{\boldsymbol{V}} = \boldsymbol{V} \cdot \text{softmax}(\boldsymbol{S}/\sqrt{d_{attn}})$

Tabla 3.2

Atención unidireccional (*Unidirectional/masked self-attention*)

También es *self-attention*, con lo que sólo se tiene una secuencia de tókenes $\boldsymbol{x}$, pero en este caso es unidireccional, por lo que se aplica el mecanismo de atención a cada token usando como contexto únicamente los tókenes anteriores (y a él mismo), es decir, $\boldsymbol{Z} = \boldsymbol{X}$ y $Mask[t_Z, t_X] := [[t_Z \leq t_X]]$.

Atención cruzada (*Cross-attention*)

Dadas dos secuencias de tókenes $\boldsymbol{x}$ y $\boldsymbol{z}$, se aplica el mecanismo de atención a todos los tókenes de la secuencia primaria $\boldsymbol{x}$ utilizando como contexto los tókenes de la secuencia $\boldsymbol{z}$. Las secuencias del *output* $\tilde{\boldsymbol{V}}$ y la primaria tienen la misma longitud l_X. Por su parte, la longitud de la secuencia de contexto puede ser distinta y se denota como l_Z. Por tanto, $\boldsymbol{X} \neq \boldsymbol{Z}$ y $Mask = 1$.

Atención *multi-head (Multi-head attention)*

En esta versión, Algoritmo 3, se considera el algoritmo de atención primario como una *attention head* y se superponen $H \in \mathbb{N}$ de ellos (con parámetros de aprendizaje diferenciados) dispuestos en paralelo para luego combinar sus *outputs*. Puede utilizarse cualquiera de los dos valores para la matriz máscara. En caso de utilizarse la versión unidireccional este bloque se denomina *masked multi-head attention*.

Conexión residual, capa de normalización y *Pointwise Feed Forward*

Se conoce como conexión residual al proceso de sumar al *output* de un proceso su *input* original. En este caso, se aplica después del proceso de *multi-headed attention*. Las conexiones residuales ayudan a entrenar a las redes neuronales haciendo que los gradientes puedan fluir a través de ellas directamente y evitando su desvanecimiento. Al hacer una suma, es necesario volver a normalizar el resultado para tener valores entre 0 y 1. Esto se consigue con una capa de normalización.

La capa de normalización, Algoritmo 6, controla explícitamente la media y la varianza de las activaciones de una red neuronal. Es común ver implementada la media cuadrática, *Root Mean Square* (RMSnorm), en

Algoritmo 3: $\tilde{\boldsymbol{V}} \longleftarrow$ AtenciónMH$(\boldsymbol{X}, \boldsymbol{Z}|\mathcal{W}, \text{Mask})$

/* Calcula una capa de *multi-head* de *self-* o *cross-attention* (*con máscara*). */

Input: $\boldsymbol{X} \in \mathbb{R}^{d_X \times l_X}$, $\boldsymbol{Z} \in \mathbb{R}^{d_Z \times l_Z}$ las representaciones vectoriales de la secuencias primaria y de contexto.

Output: $\tilde{\boldsymbol{V}} \in \mathbb{R}^{d_{out} \times l_X}$ la representación vectorial conjunta de los tókenes en $\boldsymbol{X}$ y su contexto $\boldsymbol{Z}$.

Hiperparámetros: H, número de *heads*.

Hiperparámetros: Mask $\in \{0,1\}^{l_Z \times l_X}$

Parámetros: $\mathcal{W}$, constituida por $\mathcal{W}^h_{qkv}$ para $h \in [H]$, a su vez formada por:
$\boldsymbol{W}^h_{\boldsymbol{q}} \in \mathbb{R}^{d_{attn} \times d_X}$, $\boldsymbol{b}^h_{\boldsymbol{q}} \in \mathbb{R}^{d_{attn}}$.
$\boldsymbol{W}^h_{\boldsymbol{k}} \in \mathbb{R}^{d_{attn} \times d_Z}$, $\boldsymbol{b}^h_{\boldsymbol{k}} \in \mathbb{R}^{d_{attn}}$.
$\boldsymbol{W}^h_{\boldsymbol{v}} \in \mathbb{R}^{d_{mid} \times d_Z}$, $\boldsymbol{b}^h_{\boldsymbol{v}} \in \mathbb{R}^{d_{mid}}$.
$\boldsymbol{W}^h_{\boldsymbol{o}} \in \mathbb{R}^{d_{out} \times H d_{mid}}$, $\boldsymbol{b}^h_{\boldsymbol{o}} \in \mathbb{R}^{d_{out}}$.

1 Para $h \in [H]$:
2 $\quad \boldsymbol{Y}^h \longleftarrow$ Atención$(\boldsymbol{X}, \boldsymbol{Z}|\boldsymbol{\mathcal{W}}^h_{\boldsymbol{qkv}}, Mask)$
3 $\boldsymbol{Y} \longleftarrow [\boldsymbol{Y}^1; \boldsymbol{Y}^2; ...; \boldsymbol{Y}^H]$
4 return $\tilde{\boldsymbol{V}} = \boldsymbol{W_o Y} + \boldsymbol{b_o} \boldsymbol{1}^T$

Tabla 3.3

muchos *transformers* por ser simple y computacionalmente eficiente. Esta normalización se correspondería a hacer que $\boldsymbol{m} = \boldsymbol{\beta} = 0$. Las capas de normalización tienen el efecto de estabilizar la red, nuevamente evitando efectos explosivos sobre los pesos. La suma normalizada de la conexión residual se introduce como

Algoritmo 4: $\tilde{\boldsymbol{e}} \longleftarrow$ capa_norm$(\boldsymbol{e}|\boldsymbol{\gamma}, \boldsymbol{\beta})$.

Input: $e \in \mathbb{R}^{d_e}$, activaciones de la red neuronal.

Output: $\tilde{\boldsymbol{e}} \in \mathbb{R}^{d_e}$, activaciones normalizadas

Parámetros: $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{d_e}$, escala y compensación de cada elemento

1 $\boldsymbol{m} \longleftarrow \sum_{i=1}^{d_e} \boldsymbol{e}[i]/d_e$

2 $v \longleftarrow \sum_{i=1}^{d_e} (\boldsymbol{e}[i] - \boldsymbol{m}^2)/d_e$

3 return $\tilde{\boldsymbol{e}} = \frac{\boldsymbol{e}-\boldsymbol{m}}{\sqrt{v}} \bigodot \boldsymbol{\gamma} + \boldsymbol{\beta}$, donde $\bigodot$ denota la multiplicación por elementos (producto Hadamart)

Tabla 3.4

input en una red neuronal con una arquitectura conocida como *Pointwise Feed Forward* que posibilita que la red general aprenda nuevas correlaciones en su mecanismo de atención. Sobre esta arquitectura se vuelve a realizar el proceso de conexión residual y normalización.

Pointwise Feed Forward

Tal y como aparece en la Figura 3.2 una red de arquitectura *Pointwise Feed Forward* (FFN) consiste en un par de capas lineales o *Multi Level Perceptrons* (MLP) unidas con una función de activación *ReLU*. Aplicadas sobre una matriz A, se puede computar una capa FFN siguiendo ec. (3.5), e introduciendo parámetros de las dimensiones adecuadas para la matriz.

$$FFN(A|\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2) = \boldsymbol{W}_2 \text{ReLU}(\boldsymbol{W}_1 A + \boldsymbol{b}_1 \boldsymbol{1}^T) + \boldsymbol{b}_2 \boldsymbol{1}^T \tag{3.5}$$

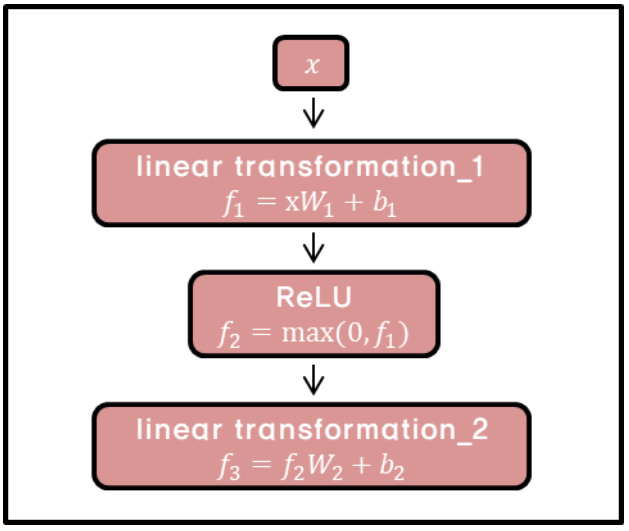

Figura 3.2: *Pointwise Feed Forward* [1]

Unembedding

Este bloque, Algoritmo 5, aprende a "desvectorizar" las representaciones de tókenes (y su contexto) obtenidas en una distribución sobre los elementos del vocabulario. Aunque en principio este bloque aprende una matriz independiente de la matriz de vectorización (*embedding matrix*), a veces se elige *ad hoc* que sea su traspuesta.

Algoritmo 5: *Unembedding*
Input: $\boldsymbol{e} \in \mathbb{R}^{d_e}$, la vectorización de un token.
Output: $\boldsymbol{p} \in \Delta(V)$, una distribución de probabilidad sobre el vocabulario.
Parámetros: $\boldsymbol{W_u} \in \mathbb{R}^{N_V \times d_e}$, la matriz *unembedding*.
1 return $\boldsymbol{p} = softmax(\boldsymbol{W_u e})$

Tabla 3.5

3.1.4. Arquitectura EDT/seq2seq original

Como se ha mencionado anteriormente, aunque en el artículo original de *Attention is all you need* se presentaba una arquitectura *Encoder-Decoder*, explícita en el Algoritmo 6, de *transformer* (EDT), BERT y GPT utilizan variaciones de esa arquitectura.

El objetivo de una estructura de *encoder-decoder* es predecir una secuencia $\boldsymbol{x}$ a partir de una secuencia $\boldsymbol{z}$. Sobre cada una de ellas se va a realizar un tipo de *self-attention*: sobre $\boldsymbol{z}$, conocida desde el principio del entrenamiento, se realiza una atención bidireccional; mientras que con $\boldsymbol{x}$ se realiza una atención unidireccional con límite variable a medida que van aumentando los tókenes predichos. Puesto que por sí solos los dos bloques de *self-attention* sobre cada una de las secuencias no tienen forma de compartir información, se añade un bloque de *cross-attention* utilizando la secuencia $\boldsymbol{x}$ como secuencia primaria y la secuencia $\boldsymbol{z}$ como secuencia de contexto. El bloque encargado de realizar la atención bidireccional sobre $\boldsymbol{z}$ se conoce como *encoder*. La atención unidireccional sobre $\boldsymbol{x}$, y la *cross-attention* con $\boldsymbol{x}$ y $\boldsymbol{z}$, se encuentran en el bloque conocido como *decoder*.

Algorítmo 6: $\boldsymbol{P} \longleftarrow$ EDTransformer$(\boldsymbol{z}, \boldsymbol{x}|\boldsymbol{\theta})$

/* Algoritmo Encoder-Decoder.
Input: $\boldsymbol{z}, \boldsymbol{x} \in V^*$, dos secuencias de tókenes de longitudes l_Z y l_X respectivamente.
Output: $\boldsymbol{P} \in (0,1)^{N_V \times l_X}$, donde la columna t-ésima de $\boldsymbol{P}$ representa $\hat{P}_\theta(x_{t+1}|x_1, ..., x_t, \boldsymbol{z})$.
Hiperparámetros: $l_{max}, L_{enc}, L_{dec}, H, d_e, d_{ffn} \in \mathbb{N}$
Parámetros: $\boldsymbol{\theta}$ incluye los siguientes parámetros:
$\boldsymbol{W_e} \in \mathbb{R}^{d_e \times N_V}, \boldsymbol{W_p} \in \mathbb{R}^{d_e \times l_{max}}$, las matrices de los *embedding* de token y *posición*.
For $l \in [L_{enc}]$:
| $\mathcal{W}_l^{enc}$, parámetros de la capa l de *multi-head attention* del *encoder*.
| $\boldsymbol{\gamma_l^1}, \boldsymbol{\beta_l^1}, \boldsymbol{\gamma_l^2}, \boldsymbol{\beta_l^2} \in \mathbb{R}^{d_e}$, dos conjuntos de parámetros de capas de normalización.
| $\boldsymbol{W}_{ffn1}^l \in \mathbb{R}^{d_{ffn} \times d_e}, \boldsymbol{b}_{ffn1}^l \in \mathbb{R}^{d_{ffn}}, \boldsymbol{W}_{ffn2}^l \in \mathbb{R}^{d_e \times d_{ffn}}, \boldsymbol{b}_{ffn2}^l \in \mathbb{R}^{d_e}$, parámetros de la FFN del *encoder* l.
For $l \in [L_{dec}]$:
| $\mathcal{W}_l^{dec}$, parámetros de la capa l de *multi-head attention* del *decoder*.
| $\mathcal{W}_l^{e/d}$, parámetros de la capa l de *multi-head cross-attention*.
| $\boldsymbol{\gamma_l^3}, \boldsymbol{\beta_l^3}, \boldsymbol{\gamma_l^4}, \boldsymbol{\beta_l^4}, \boldsymbol{\gamma_l^5}, \boldsymbol{\beta_l^5} \in \mathbb{R}^{d_e}$, tres conjuntos de parámetros de capas de normalización.
| $\boldsymbol{W}_{ffn3}^l \in \mathbb{R}^{d_{ffn} \times d_e}, \boldsymbol{b}_{ffn3}^l \in \mathbb{R}^{d_{ffn}}, \boldsymbol{W}_{ffn4}^l \in \mathbb{R}^{d_e \times d_{ffn}}, \boldsymbol{b}_{ffn4}^l \in \mathbb{R}^{d_e}$, parámetros de la FFN del *decoder* l.
$\boldsymbol{W_u} \in \mathbb{R}^{n_V \times d_e}$, la matriz *embedding*. |
/* *Encoder: m-h. self-attention* $\boldsymbol{z}$.
$l_Z \longleftarrow length(\boldsymbol{z})$
for $t \in [l_Z] : \boldsymbol{e}_t \longleftarrow \boldsymbol{W_e}[:, z_t] + \boldsymbol{W_p}[:, t]$
$\boldsymbol{Z} \longleftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_{l_Z}]$
for $l = 1, 2, ..., L_{enc}$ **do**:
| $\boldsymbol{Z} \longleftarrow \boldsymbol{Z} +$ AtenciónMH$(\boldsymbol{Z}|\mathcal{W}_l^{enc}, \text{Mask} \equiv 1)$
| for $t \in [l_Z] : \boldsymbol{Z}[:, t] \longleftarrow$ capa_norm$(\boldsymbol{Z}[:, t]|\boldsymbol{\gamma}_l^1, \boldsymbol{\beta}_l^1)$
| $\boldsymbol{Z} \longleftarrow \boldsymbol{Z} + FFN(\boldsymbol{Z}|\boldsymbol{W}_{ffn1}^l, \boldsymbol{b}_{ffn1}^l, \boldsymbol{W}_{ffn2}^l, \boldsymbol{b}_{ffn2}^l)$
| for $t \in [l_Z] : \boldsymbol{Z}[:, t] \longleftarrow$ capa_norm$(\boldsymbol{Z}[:, t]|\boldsymbol{\gamma}_l^2, \boldsymbol{\beta}_l^2)$
end
/* *Decoder: m-h. self-attention* con $\boldsymbol{x}$; *m-h. cross-attention* con $\boldsymbol{x}, \boldsymbol{z}$.
$l_x \longleftarrow length(\boldsymbol{x})$
for $t \in [l_Z] : \boldsymbol{e}_t \longleftarrow \boldsymbol{W_e}[:, x_t] + \boldsymbol{W_p}[:, t]$
$\boldsymbol{X} \longleftarrow [\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_{l_x}]$
for $l = 1, 2, ..., L_{dec}$ **do**:
| $\boldsymbol{X} \longleftarrow \boldsymbol{X} +$ AtenciónMH$(\boldsymbol{X}|\mathcal{W}_l^{enc}, \text{Mask}[t, t'] \equiv [[t \leq t']])$
| for $t \in [l_X] : \boldsymbol{X}[:, t] \longleftarrow$ capa_norm$(\boldsymbol{X}[:, t]|\boldsymbol{\gamma}_l^3, \boldsymbol{\beta}_l^3)$
| $\boldsymbol{Z} \longleftarrow \boldsymbol{X} +$ AtenciónMH$(\boldsymbol{X}|\mathcal{W}_l^{e/d}, \text{Mask} \equiv 1)$
| for $t \in [l_X] : \boldsymbol{X}[:, t] \longleftarrow$ capa_norm$(\boldsymbol{X}[:, t]|\boldsymbol{\gamma}_l^4, \boldsymbol{\beta}_l^4)$
| $\boldsymbol{X} \longleftarrow \boldsymbol{X} + FFN(\boldsymbol{X}|\boldsymbol{W}_{ffn3}^l, \boldsymbol{b}_{ffn3}^l, \boldsymbol{W}_{ffn4}^l, \boldsymbol{b}_{ffn4}^l)$
| for $t \in [l_X] : \boldsymbol{X}[:, t] \longleftarrow$ capa_norm$(\boldsymbol{X}[:, t]|\boldsymbol{\gamma}_l^5, \boldsymbol{\beta}_l^5)$
end
/* Inferir las probabilidades condicionales y finalizar.
return $\boldsymbol{P} =$ softmax$(\boldsymbol{W_u X})$

Para entenderlo mejor, se va a ver paso a paso con una tarea ficticia. Esta no es una tarea sobre la que se entrene ningún modelo de la familia de BERT (aunque está basada en ella) o GPT.

Considérese la oración "este buscador es recomendable para ..... tiempo", donde se ha enmascarado/escondido la subsecuencia de tókenes "perder el". Este enmascaramiento no se realiza con la matriz *Mask*, sino que durante el entrenamiento cada token de la subsecuencia es sustituido por un *dummy token* denominado *mask_token* (este proceso es análogo al entrenamiento de BERT, pero como se verá más adelante, con BERT cada token en el *input* se reemplaza por un *mask_token* en base a una probabilidad p_{mask} durante el entrenamiento, sin formar subsecuencias de tókenes). En esta tarea, la secuencia de

contexto en el bloque de *cross-attention* $\boldsymbol{z}$ sería la oración sin máscara, y la secuencia primaria $\boldsymbol{x}$ la subsecuencia de tókenes a predecir. Para este ejercicio mental supongamos que la tokenización es por palabras, de manera que se han escondido dos tókenes. Incluso para una persona pudiera llegar a ser difícil de resolver, pero no imposible, puesto que "perder el tiempo" es una expresión bastante habitual y, por lo tanto, el grado de atención entre las palabras "perder", "el" y "tiempo" es alto.

Input embedding

En primer lugar, se detectan y se vectorizan (con valores continuos) las palabras en nuestra oración por separado, proceso conocido como *token embeddding*. Esto no es un proceso trivial, los vectores que recogen cada palabra tienen información sobre su sentido semántico, pero desafortunadamente no en el contexto de la frase en particular sobre la que aparecen. En el caso de las palabras polisémicas esto es un problema bastante importante:

"Me senté en un banco antes de ir."

"Me senté antes de ir al banco."

Los mecanismos de atención van a dar cuenta precisamente de esto mismo. Para ello, se añade la información de cada una de las posiciones de las palabras en la oración al vector correspondiente, es decir, el *positional enmbedding*.

Se denomina *input embedding* a la combinación del *token* y el *positional embedding*, y el formalismo es el que se explicó anteriormente.

Bloque del *encoder*

El *encoder* es el encargado de generar un representación continua de $\boldsymbol{z}$, es decir, "Este buscador es recomendable para ..... tiempo". Para ello, recibe como *input* los *input embeddings* de cada token en la secuencia y realiza *multi-head self-attention* sobre ella, y se le añade al bloque una conexión residual y una capa de normalización. El resultado se pasa por un bloque de *Pointwise Feed Forward*, con su correspondiente conexión residual y normalización. Se suelen solapar N_x capas con la estructura señalada en el recuadro izquierdo de la Figura 3.1.

Bloque del *decoder*

El *decoder* se encarga de predecir la secuencia $\boldsymbol{x}$, idealmente la subsecuencia de tókenes "perder el". Para ello, predice en orden (secuencialmente) cada token en la secuencia, realizando *masked multi-head attention* con límite en el token sobre el que se realiza la atención en cada caso. Luego, realiza *multi-head cross attention* utilizando las representaciones de $\boldsymbol{z}$ obtenidas por el *encoder* como contexto, y las representaciones obtenidas por el bloque de atención unidireccional como representaciones de la secuencia primaria. Cada bloque de atención incluye conexión residual y normalización. Después del último bloque de atención el resultado se pasa por un bloque de *pointwise feed forward*, también con su correspondiente conexión residual y normalización. Tal y como sucede con el *encoder*, el *decoder* puede solapar N_x capas con la misma arquitectura.

Por último, se hace pasar al resultado por una capa lineal que da acceso a un clasificador. El número de etiquetas de este clasificador será el número de palabras de nuestro vocabulario (N_V). Al final del proceso el modelo asignará un valor a cada palabra y una capa final de *softmax* ayudará al modelo a diferenciar

las más probables durante el entrenamiento. El resultado final del proceso viene dado por el índice de la palabra más probable.

En el ejemplo que estamos manejando el modelo debería auto-completar con "perder el". Para la segunda palabra, "el", el *decoder* utilizará las representaciones de "Este buscador es recomendable para ..... tiempo" obtenidas por el *encoder*, y la representación de la primera palabra predicha de la subsecuencia enmascarada, idealmente: "perder".

3.1.5. Otras Arquitecturas

En la práctica, la tarea del ejemplo no es exactamente la que se utiliza para entrenar estos modelos. En el caso de BERT, solo se extrae una palabra de cada oración para generar los *inputs*, por tanto no es necesaria una capa de *masked multi-head attention* que dé cuenta de los *outputs* previos (sólo hay que predecir una palabra). De esta forma, el *trasformer* acaba siendo sólo una concatenación de *encoders*.

Por otro lado, GPT se entrena con la tarea de predecir la siguiente palabra en una oración. En cada iteración, la frase generada pasa a ser el nuevo *input*. En este caso sí que es necesario una capa de *masked multi-head attention*, pero en vez de introducir sólo las palabras predichas como *outputs*, se introduce la frase al completo. De esta forma es posible prescindir del bloque del *encoder*.

Aunque la diferencia principal entre ambas arquitecturas es la matriz *máscara*, también utilizan diferentes funciones de activación y las capas de normalización se disponen de manera distinta.

***Encoder-only transformer*: BERT, RoBERTa**

Además de implementar únicamente el módulo del *encoder*, BERT también utiliza la no-linealidad de GELU en vez de ReLU:

$$GELU(x) = x \cdot \boldsymbol{P}_{X\sim\mathcal{N}(0,1)}[X < x] \tag{3.6}$$

Por su parte, la diferencia entre BERT y RoBERTa está en que el segundo realiza el enmascaramiento, de forma dinámica (y no aleatoria) [15]. En ambos se entrenan *transformers* de distintos tamaños, siendo los dos más comunes el tamaño *base* y el tamaño *large*.

- *BERT base*: 12 capas (bloques de *transformer*), 12 *attention-heads* y 110 millones de parámetros.
- *BERT large*: 24 capas, 16 *attention-heads* y 340 millones de parámetros.

El uso original de BERT era aprender representaciones útiles de tókenes con su contexto, que después podían ser adaptadas (*fine tuned*) a distintas tareas de NLP.

Sentence BERT

La forma de adaptar BERT más importante para este trabajo tiene como objetivo la similitud semántica entre textos, *Semantic Textual Similarity* (STC), y más concretamente entre oraciones, *sentence similarity*. BERT de base genera representaciones para cada tókenes y, por lo tanto, comparar dos oraciones no es trivial.

La forma original de hacerlo es introducir ambas oraciones en una misma red de *encoder-only* pre-entrenada y alterarla de manera que el *output* pase por un *clasificador* que asigne un valor entre 0 y 1 indicando la similitud. Esta solución se conoce como *cross-encoder*. Es importante darse cuenta de que un *cross-encoder* no genera representaciones, en el sentido de que no se le puede pasar una única oración y

esperar que devuelva un vector. Por tanto, para cada pareja de oraciones distinta es necesario realizar las representaciones de las dos, aunque alguna de las oraciones ya se haya utilizado previamente en otra comparación. Esto hace que un *cross-encoder* sea computacionalmente muy caro.

Para solventar esto, *Sentence-BERT* (SBERT) entrena una red neuronal *siamesa/triple*, que introduce cada oración en una red BERT distinta y que genera representaciones para cada una de ellas, ambas con las mismas dimensiones para hacer posible su comparación [17]. Este tipo de modelos se suelen conocer como *bi-encoders* (no confundir con *bidirectional encoders*). De esta forma se consigue un modelo entrenado en realizar representaciones de oraciones, y como la representación de cada oración sólo se calcula una vez, se reduce muy notablemente el coste computacional con respecto al de un *cross-encoder*. Para entrenar estas redes neuronales se utilizan distintos conjuntos de datos y distintas funciones objetivo en función de cuál sea exactamente la tarea de STC que se requiera (*sentence similarity* no es la única).

Las tres funciones objetivos utilizadas son:

- **Función de clasificación:**

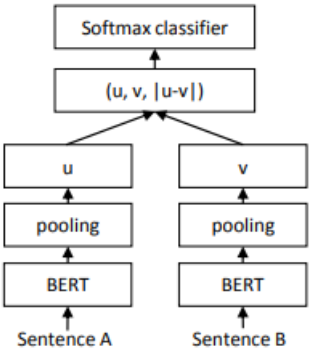

Figure 1: SBERT architecture with classification objective function, e.g., for fine-tuning on SNLI dataset. The two BERT networks have tied weights (siamese network structure).

Figura 3.3: Arquitectura SBERT con función objetivo de clasificación. [17]

Se calculan las representaciones de los tókenes de dos oraciones A y B utilizando BERT. Las representaciones $\boldsymbol{U}'$ y $\boldsymbol{V}'$ obtenidas se pasan por una capa que las reduce a dos vectores $\boldsymbol{u}$ y $\boldsymbol{v}$ de la misma dimensión, denominada capa de *pooling*. Se concatenan $\boldsymbol{u}$, $\boldsymbol{v}$ y la diferencia elemento a elemento $|\boldsymbol{u} - \boldsymbol{v}|$ para finalmente multiplicar el resultado por la matriz de parámetros de entrenamiento $\boldsymbol{W}_t \in \mathbb{R}^{3n \times d_k}$, donde n es la dimensión de $\boldsymbol{u}$ y $\boldsymbol{v}$ y d_k el número de etiquetas. Finalmente, se aplica una función *softmax* al resultado. Esta arquitectura viene reflejada en la Figura 3.3 y en el pseudocódigo del Algoritmo 7. En este algoritmo se hace referencia a EInfrence$(\boldsymbol{x}_n, \hat{\boldsymbol{\theta}})$, que recibe como *input* una secuencia de tókenes $\boldsymbol{x}_n$ y los parámetros entrenados $\hat{\boldsymbol{\theta}}$ de BERT y devuelve las representaciones $\boldsymbol{U}'$ de sus tókenes. También se hace referencia a la capa de *pooling* que recibe estas representaciones $\boldsymbol{U}'$ y devuelve un vector $\boldsymbol{u}$.

- **Función de regresión:** Mismo procedimiento pero, en vez de concatenar $\boldsymbol{u}$ y $\boldsymbol{v}$ y aplicar un *softmax*, se calcula el producto escalar entre ambos vectores. En esta disposición, la capa de *pooling* demuestra ser fundamental para obtener unos buenos resultados.

- **Función *triplet*:** Dada una oración ancla a, una oración positiva p y una negativa n, la función de

Algoritmo 7: $\hat{\boldsymbol{W}}_t \longleftarrow$ SBERT_clas_training$(\boldsymbol{z}^1, ..., \boldsymbol{z}^{n_{data}}, \boldsymbol{x}^1, ..., \boldsymbol{x}^{n_{data}}, \hat{\boldsymbol{\theta}}, K, \boldsymbol{W}_t)$

/* Entrenamiento de SBERT con objetivo clasificación.
Input: K conjunto de etiquetas.
Input: $\boldsymbol{W_t} \in \mathbb{R}^{3n \times d_k}, n \in \mathbb{N}, d_k = |K|$ los parámetros de la red neuronal siamesa a entrenar.
Input: $\hat{\boldsymbol{\theta}}$ los parámetros entrenados de BERT.
Input: $\{(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}, k)\}_{n=1}^{n_{data}}$ un conjunto de secuencias emparejadas con etiqueta $k \in K$.
Output: $\hat{\boldsymbol{W}}_t$ los parámetros entrenados de la red siamesa.
Hiperparámetros: $N_{epochs} \in \mathbb{N}, \eta \in (0, \infty)$
1 for $i = 1, 2, ..., N_{epochs}$ **do**:
2 | **for** $n = 1, 2, ..., n_{data}$ **do**:
3 | | $\boldsymbol{u} \longleftarrow pooling(\text{EInference}(\boldsymbol{x}^{(n)}, \hat{\boldsymbol{\theta}}))$
4 | | $\boldsymbol{v} \longleftarrow pooling(\text{EInference}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{\theta}}))$
4 | | $\boldsymbol{o} \longleftarrow softmax(\boldsymbol{W}_t(\boldsymbol{u}, \boldsymbol{v}, |\boldsymbol{u} - \boldsymbol{v}|))$
5 | | loss $\longleftarrow$ Entropía cruzada entre $\boldsymbol{o}$ y $\boldsymbol{W}_t$.
6 | | $\boldsymbol{W}_t \longleftarrow \boldsymbol{W}_t - \eta \cdot \nabla \text{loss}(\boldsymbol{W}_t)$
7 | **end**
8 end
9 return $\hat{\boldsymbol{W}}_t = \boldsymbol{W}_t$

pérdida triple entrena a la red de manera que la distancia entre a y p es menor que la distancia entre a y n. Formalmente se minimiza la siguiente función de pérdida:

$$\max(||\boldsymbol{a} - \boldsymbol{p}|| - ||\boldsymbol{a} - \boldsymbol{n}|| + \epsilon, 0)$$

donde $\boldsymbol{a}, \boldsymbol{p}$ y $\boldsymbol{n}$ son los *embeddings* de a, p y n respectivamente, $||\cdot||$ una métrica de distancia y ϵ un margen. Este margen se ocupa de que $\boldsymbol{p}$ esté al menos ϵ más cerca de $\boldsymbol{a}$ que $\boldsymbol{n}$.

Decoder-only transformer: GPT-2, GPT-3, Gopher

Estos modelos únicamente implementan la parte del *decoder*. Gopher es el modelo de DeepMind análogo a GPT. GPT-2 y GPT-3 son idénticos, salvo que la versión 3 es mucho más grande y reemplaza la antención *densa* por atención *sparse*, es decir, sólo se utiliza un subconjunto de todo el contexto para cada token. Por su parte, Gopher solo se diferencia de GPT-2 en utilizar distintos *positional embeddings* y en utilizar RMSnorm$(m = \boldsymbol{\beta} = 0)$ como capa de normalización.

Además de eso, una diferencia práctica fundamental con respecto a los modelos *encoder-only* como BERT es que GPT-3 no necesita un *fine-tunning* para cada tarea de NLP, sino que simplemente con un ejemplo es capaz de reproducirla. A cambio, su tamaño es mucho menos manejable que los *encoders* tradicionales.

Multi-domain decoder-only transformer: Gato

Implementa únicamente el módulo del *decoder*.

3.1.6. Entrenamiento y predicciones con *transformers*

Nuevamente, solo se presenta el entrenamiento para un *transformer* completo. Een el Algoritmo 8 se da una descripción detallada del mismo [2].

Algorítmo 8: $\boldsymbol{P} \longleftarrow \text{EDTraining}(\boldsymbol{z}^1, ..., \boldsymbol{z}^{n_{data}}, \boldsymbol{x}^1, ..., \boldsymbol{x}^{n_{data}}, \theta)$

/* Entrenamiento de un modelo seq2seq.
Input: $\{(z^n, x^n)\}_{n=1}^{n_{data}}$ un conjunto de parejas de secuencias.
Output: $\hat{\boldsymbol{\theta}}$ los parámetros entrenados.
Input:$\boldsymbol{\theta}$ parámetros iniciales del *transformer*.
Output: $\hat{\boldsymbol{\theta}}$ los parámetros entrenados.
Hiperparámetros: $N_{epochs} \in \mathbb{N}, \eta \in (0, \infty)$
1 for $i = 1, 2, ..., N_{epochs}$ **do**:
2 | **for** $n = 1, 2, ..., n_{data}$ **do**:
3 | | $l \longleftarrow \text{length}(x_n)$
4 | | $\boldsymbol{P(\theta)} \longleftarrow \text{EDTransformer}(\boldsymbol{z}^{(n)}, \boldsymbol{x}^{(n)} | \boldsymbol{\theta})$
5 | | $\text{loss}(\boldsymbol{\theta}) = -\sum_{t=1}^{l-1} \log P(\boldsymbol{\theta})[x_{t+1}^{(n)}, t]$
6 | | $\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} - \eta \cdot \nabla \text{loss}(\boldsymbol{\theta})$
7 | **end**
8 end
9 return $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}$

[2] Aunque en él la optimización se realiza con un descenso de gradiente, en la práctica se suelen utilizar algoritmos más refinados como *Adam*[18].

Capítulo 4

Una aplicación práctica: la búsqueda de información

Una de las aplicaciones prácticas del procesado de lenguaje natural es la posibilidad de realizar búsquedas semánticas. En particular, el problema que se me presentó durante las prácticas en Incentro fue encontrar, en una base de datos, aquellos atributos más "relevantes" dado un cierto producto. Por ejemplo, para el producto "Aceite de oliva virgen extra", los atributos "aceite vegetal", "aceite extra virgen" y "aceite de oliva" se consideran relevantes, mientras que los atributos "extra picante´´ o "aceites de manos", no. De hecho, su relevancia es puramente subjetiva.

4.1 Modelos de búsqueda de información

4.1.1. Modelo tradicional TF-IDF

Las técnicas clásicas de búsqueda y recuperación de la información no se fundamentaban en utilizar el significado semántico de los tókenes/palabras (la tokenización habitual en estos algoritmos es por palabras) u oraciones, simplemente realizaban un conteo inteligente de las palabras de una búsqueda (*query*) sobre unos documentos (*corpus*) y mostraban el documento que mejor puntuase con respecto a ese conteo.

Uno de las formas de puntuar más habituales en este tipo de técnicas es la conocida como *term frequency - inverse document frequency (tf-idf)*. Se fundamenta en la idea de que si una palabra de la *query* se repite mucho en un documento del *corpus*, ese documento obtiene una puntuación alta para salir como un posible resultado, pero solo si esa palabra es representativa de la intención de la *query*. Esto se pondera contando cuántas veces sale la palabra en el *corpus* al completo: si sale en muchos de los documentos, la palabra se considera demasiado general (una preposición o una conjunción por ejemplo) como para puntuar tan positivamente a los documentos que la contengan.

Este proceso se hace para cada palabra de la *query* y al final se tiene una puntuación para cada documento en el *corpus*. En realidad, en este tipo de algoritmos no se computa una puntuación únicamente contando, sino que se le suele aplicar alguna función para mejorar los resultados, por ejemplo la logarítmica. Veamos cómo formalizar estos conceptos con el algoritmo de *tf-idf* utilizado en este trabajo: *Okapi BM25*.

Okapi BM25

Dada una consulta Q, que contiene las palabras clave $q_1, ..., q_n$, el valor de relevancia viene dado por la ec. (4.1):

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \tag{4.1}$$

donde $f(q_i, D)$ es la frecuencia de aparición en el documento D del token q_i (en este caso palabra) que aparece en la consulta Q, $|D|$ es el número de palabras del documento D, y $avgdl$ es el número de palabras medio de los documentos del *corpus*. k_1 y b son parámetros que permiten ajustar la función a las características concretas del *corpus*, aunque por lo general se les dan los valores $k_1 = 2{,}0$ o $k_1 = 1{,}2$ y $b = 0{,}75$, que han probado ser eficientes tras varios años de experimentos.

El término de $IDF(q_i)$ representa el peso del *inverse document frequency* de cada palabra q_i de la *query* Q. En BM25 se calcula con la ec. (4.2):

$$IDF(q_i) = \log \frac{N - n(q_i) + 0{,}5}{n(q_i) + 0{,}5} \tag{4.2}$$

donde N es el número de documentos en el *corpus* y $n(q_i)$ es el número de documentos que contienen la palabra q_i.

4.1.2. Búsqueda semántica con *transformers*

Aunque son útiles, las técnicas de búsqueda de información tradicionales como las basadas en *tf-idf* tienen una lacra fundamental: carecen de cualquier tipo de sentido semántico. Es por eso que, entendiendo los términos de *query* y *corpus* en el sentido de TF-IDF, en lo que va a consistir la aplicación de *transformers* a este campo es en el uso de *cross-encoders* o *bi-encoders* para calcular la similitud semántica entre la *query* y cada documento del *corpus*. En particular, se implementa un modelo *bi-encoder* multi-lenguaje entrenado con la metodología de *Sentence-BERT* desarrollada por *Hugging Face* [19] y explicada en el capítulo anterior.

4.2 Búsqueda de atributos en GPC

El problema en particular consiste en asignar productos para hoteles buscando los atributos más relevante para cada producto según la base datos de *Global Product Classification*(GPC). Los atributos de esta base de datos se pueden acceder libremente en formato *.xml* y *.json*. Este estándar de GPC tiene una estructura de bloque y sub-bloque con el esquema de la Figura 4.3

Tal y como se muestra en la Figura 4.3, GPC divide los productos en segmentos, y los subdivide sucesivamente en familia, clase, *brick*, tipo de atributo y valor de atributo. Cada una de estas secciones consta de código, título y descripción. En principio, al cliente le interesa poder pasar productos y ser capaz de encontrar los *bricks* más adecuados. Para ello, la idea es utilizar el nombre del producto como una *query* en un sistema de búsqueda y recuperación de información donde el *corpus* son los títulos de las categorías de GPC. Idealmente solo sería necesario comparar los productos con los *bricks*, pero en la práctica lo más parecido a lo nombres de los productos son los títulos de los valores de los atributos, el último escalón del esquema de GPC. Por tanto, las arquitecturas de búsqueda implementadas buscan en

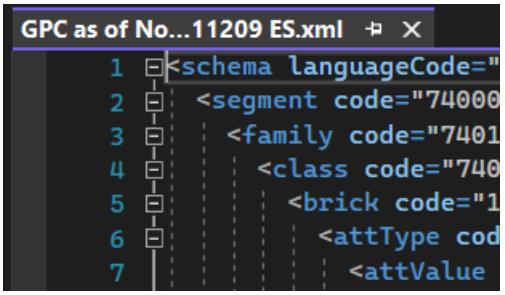
GPC as of No...11209 ES.xml
<schema languageCode="
<segment code="74000
<family code="7401
<class code="740
<brick code="1
<attType cod
<attValue

Figura 4.1: Clasificación según GPC

primer lugar los valores de los atributos que más se parezcan al producto, y luego encuentran el *brick* que, conteniendo esos atributo, sea de más relevancia finalmente para el producto.

De cara al problema clásico de búsqueda y recuperación de información, se va a realizar una comparación entre distintos algoritmos solo en la búsqueda de los valores de atributos. La elección del *brick* no tiene tanto valor académico pues es algo muy particular a este esquema de GPC; hacer *queries* directamente sobre una base de datos de forma semántica, sí.

4.2.1. Arquitecturas

BM25

Se implementa el algoritmo de Okapi BM25 explicado en la sección anterior.

SBERT

Se implementa *Sentence-BERT* tal y como se explica en la sección anterior y en el Capítulo 3.

Intersección BM25 con SBERT

La lógica es sencilla: utilizar BM25 y SBERT paralelamente y luego interseccionar los valores de atributos resultantes. En caso de que no haya resultados coincidentes, quedarse con los de SBERT. Esta decisión esta justificada porque la búsqueda con SBERT es semántica, y la no-obtención de resultados generalmente se debía a que las palabras de los productos no estaban entre las palabras de los valores de atributos, por lo que la técnica de TF-IDF no encontraba coincidencias.

BM25 refinado por SBERT

Esta es la arquitectura sugerida por *HuggingFace* para el uso de *SBERT* en búsquedas semánticas. Consiste en implementar un algoritmo de búsqueda estilo BM25 para la obtención de los k resultados más relevantes, y luego en seleccionar el orden de los resultados utilizando *SBERT*. De esta forma se reduce notablemente el número de productos escalares a realizar con la clasificación de cada producto.

4.2.2. Limpieza con `Spacy`

Con el objetivo de mejorar el rendimiento de los anteriores modelos se ha utilizado una biblioteca de lenguaje natural muy popular: `Spacy` [20]. Con ella se ha pretendido extraer las relaciones entre los tókenes de los sintagmas nominales que aportan el sentido semántico adecuado para su clasificación.

4.2.3. Back-Translation y Fine-Tunning

Implementado *ad hoc*, SBERT no tiene en cuenta el vocabulario en específico que se está manejando en el proyecto en particular donde se utiliza. Si se dispone de datos etiquetados, en este caso de productos ya etiquetados con el estándar GPC, se puede entrenar a SBERT a modificar sus representaciones continuas de manera que se adapten a lo que ve con los productos etiquetados. Para ello, la función de perdida que mejor ha demostrado funcionar es *Multiple Negative Loss*. A este proceso de modificación para adaptar las vectorizaciones al problema en particular al que se enfrenta el modelo sin re-entrenarlo por completo se le conoce por *fine-tunning*.

En este proyecto no se disponía de estas etiquetas, problema conocido como *cold-start*. Uno de los intentos para generar datos etiquetados ha consistido en la técnica de *Back-Translation*. Se ha partido de las listas de *bricks* y de valores de atributos y se han traducido del español al inglés y al francés, y de estos idiomas de vuelta al español. Debido a la aleatoriedad en las traducciones hay pequeños cambios en las traducciones que se pueden utilizar como nuevos datos con etiqueta el nombre original. Realmente el proceso está diseñado para un *corpus* con una mayor riqueza semántica, el problema tanto de las listas de GPC como de los productos a clasificar es que en su mayoría son sintagmas nominales. Los resultados han sido pobres y no se han incluido en la sección de resultados.

4.3 Montaje en producción

La lógica del motor de búsqueda es una cosa, pero la implementación en un entorno de producción es un desafío en sí mismo. En primer lugar, la idea del cliente era pasar una lista de productos (alrededor de 600.000) y buscar para todo los productos los atributos más relevantes de acuerdo a GPC. Con la búsqueda de atributos para cada producto mediante cualquiera de las arquitecturas (menos con BM25), se debían realizar las vectorizaciones con SBERT de ese producto y de varios atributos asociados. Además, se debían calcular los productos escalares correspondientes (en el caso de la arquitectura de intersección como mínimo trece mil, uno por cada valor de atributo único). El volumen de operaciones y de datos presentaba el primero de los problemas en la implementación: la *paralelización* en las ejecuciones. En primer lugar es necesario *vectorizar* el código, esto es, expresar las operaciones como productos escalares y matriciales. En segundo lugar es necesario contar con las piezas de *hardware* que posibilitan físicamente estas operaciones: las GPUs, las *Tensor Processing Units* (TPUs) o más recientemente, los ordenadores analógicos.

El segundo problema consistía en que además de la carga inicial de productos, el cliente tenía intención de sumar proveedores de productos a su plataforma de venta, añadiéndolos a su lista general de productos con atributos de GPC asociados por relevancia. De esta forma, si la cartera de proveedores crecía lo suficiente, el *hardware* encargado del almacenamiento de los datos podría quedarse obsoleto. El cliente buscaba lo que en el mundo informático se conoce como escalabilidad.

La solución más común en estos últimos años es recurrir a los servicios de un tercero con el *hardware* necesario para garantizar la paralelización en las ejecuciones y la escalabilidad: los servicios en la nuble o

servicios *cloud*. En este caso los servicios elegidos han sido los de *Google Cloud*.

4.3.1. Apache Beam en Google Cloud: Dataflow

La idea fundamental al montar un proceso escalable y computacionalmente eficiente en *cloud* es que almacenamiento y computación son servicios separados. Más mundanalmente, esto también se traduce en la factura al final de mes: los recursos computacionalmente son notablemente más caros, así que mejor solo levantarlos cuando se necesiten (ambos se pagan por horas).

En la jerga informática actual, los servicios locales tradicionales (los que no son *cloud*) se conocen como *on premise*. La administración del *hardware* para garantizar las paralelizaciones y la escalabilidad ha sido (y es) un problema administrado por equipos especializados, la diferencia es que en *cloud* viene incluido en el precio. Ha habido disintas tecnologías que se han utilizado para esto: Hadoop, Spark, Hive, Apache Beam, etc. El servicio recomendado en Google Cloud para un proceso como el de este proyecto está basado en Apache Beam y recibe el nombre de Dataflow.

En Dataflow [21] (y en Apache Beam en general), los procesos de ingesta, transformación y salida de datos transformados se conocen como *pipelines*. Por ejemplo, en este proyecto se ha realidad una *pipeline* donde se leen los productos del lugar donde están almacenados, se transforman paralelamente añadiéndoles una etiqueta de GPC, y se vuelven a escribir en un lugar de almacenamiento.

El lugar de almacenamiento en Google Cloud que garantiza escalabilidad y que mejor se adapta al volumen de datos del proyecto es Cloud Storage.

4.3.2. Programación en GPUs: Tensoflow, PyTorch, Cupy, Numba y PyCuda

Para este proyecto la administración del almacenamiento no ha sido demasiado complicada, sin embargo, ha sido interesante enfrentarse a las complicaciones de la paralelización de código en general y en *Machine/Deep Learning* en particular. La pieza de *harware* elegida para las ejecuciones ha sido la GPU (Google también ofrece TPUs, pero están más orientadas al entrenamiento de grandes redes neuronales profundas, paralelizar las vectorizaciones, productos escalares y búsquedas de máximos de este proyecto no eran necesarias). Existe una disciplina de la programación específicamente enfocada en la programación sobre GPU. Pero, ¿a qué se refiere esto?

Para entender verdaderamente que significa programar sobre el *hardware* específico de una GPU es necesario salirse un poco de la programación más superficial y entender que es lo que pasa cuando se programa algo a nivel de *hardware*. Pero esto no es un trabajo informático, así que he decidido dar una explicación sencilla (aunque posiblemente no todo lo correcta que debería).

Tradicionalmente hay tres componentes principales que hay que tener en cuenta cuando se programa algo a nivel de producción: disco, CPU (*Central Processing Unit*) y RAM (*Random Access Memory)*. A grandes rasgos, la CPU es la pieza encargada de realizar las operaciones y el juego de la eficiencia consiste en decidir donde se realiza el almacenamiento de los componentes de las operaciones. Guardar cosas en disco no ralentiza las computaciones por si mismo, pero a cambio es más lento a la hora de acceder a la información. En el caso de *cloud*, además, disco es un servicio externo, así que este problema se acentúa. Por su parte, guardar las cosas en memoria facilita su acceso para las computaciones, pero si se sobrecarga las ralentiza por ser parte del mismo *hardware* donde se realizan.

Cuando se añade una GPU a la ecuación, es necesario entender que cada GPU viene asociada no con una, sino con varias RAM. Por fortuna, al nivel de implementación que se va a a realizar en este trabajo se puede pensar como una sola RAM asociada a la GPU. El juego consiste en decidir dónde realizar el

almacenamiento, pero encima teniendo en cuenta de que las cosas almacenadas en la RAM de la CPU solo pueden computarse con la CPU, y de que las cosas almacenadas en la GPU solo pueden computarse con la GPU. Además, a nivel tecnológico las RAMs implementadas en las GPUs son más modernas y rápidas que las de las CPUs.

En la Figura 4.2 se puede apreciar la diferencia entre el número de núcleos (*cores*) y cachés (*cache*) de las CPUs y GPUs respectivamente.

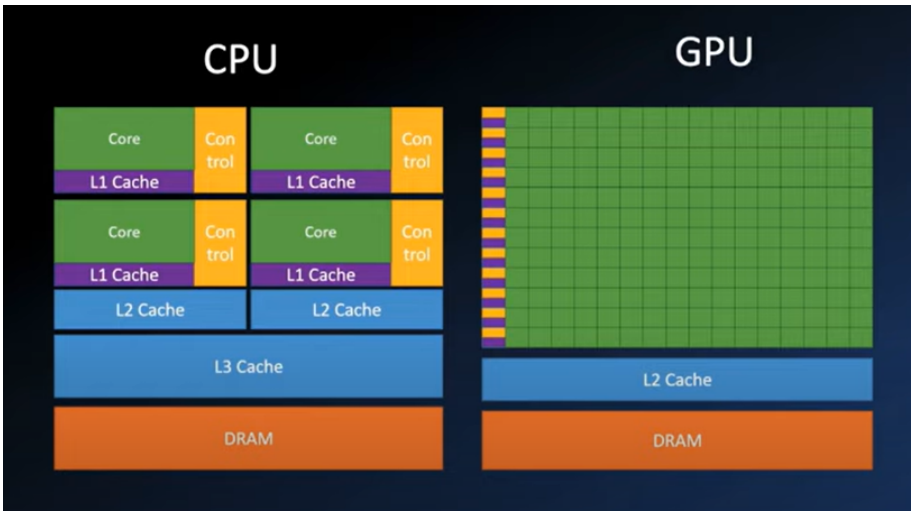

Figura 4.2: CPU vs GPU

El lenguaje por excelencia para la paralelización de ejecuciones sobre una GPU es C++, y más concretamente lo normal es CUDA. Por suerte, existen varias implementaciones en Python de distintos niveles de profundidad.

- `PyTorch` y `Tensorflow` son librerías desarrolladas por Facebook y Google respectivamente para el entrenamiento de modelos de *Machine/Deep Learning*. Pero además de venir con una amplia batería de algoritmos implementados, vienen con funciones y métodos específicamente diseñados para correr sobre GPU/TPUs.

- `CuPy` es una librería más general que las dos anteriores. Pretende ser un reflejo de `NumPy` pero implementada sobre GPUs, además de un puente entre estas y las CPUs. Su uso en principio no está tan restringido al Aprendizaje Automático. Incluye, por ejemplo, productos matriciales, escalares, etcétera, ya optimizados para su realización sobre GPU.

- `Numba` es ya una librería que permite generar ya funciones propias sobre la GPU. En general su uso es recomendable solo si la función en particular no está ya implementada en las librerías anteriores.

- `PyCuda`: es la versión más profunda de programación sobre GPU en lenguaje Python.

4.4 Resultados

Para la comparativa de los resultados de los modelos se ha elegido una muestra de 20 productos al azar y se han buscados lo valores de atributos más relevantes con las distintas arquitecturas.

4.4.1. Métricas

Es importante señalar que la elección de la relevancia de los atributos para cada producto es subjetiva, y que la naturaleza del problema dificulta la aplicación de métricas estadísticas como sesgo, varianza, F1, etc. Es por eso que se han definido dos métricas para poder comparar las arquitecturas de búsqueda implementadas.

Métrica 1

Se consideran los primeros diez resultados de posibles valores de atributos dados por cada uno de los modelos, y se valora únicamente el que mejor responda (si es que lo hubiera y siempre según el entendimiento del evaluador) al significado del producto. De existir, se puntúa con $\frac{1}{posición}$, donde la *posición* es la posición en la cuál aparecía el atributo en la lista. De esta forma, se pondera que el modelo de búsqueda sea mejor según dé el valor de atributo correcto en las primeras posiciones. Se hace para cada uno de los 20 productos y se suman lo resultados, obteniendo así una puntuación para cada modelo. Puesto que son 20 productos, la mejor puntuación que se puede obtener con esta métrica es 20 puntos.

$$\text{Métrica 1} = \sum_{i=1}^{20} \frac{1}{\text{posición de atributo correcto para producto i}}$$

Métrica 2

En este caso, se consideran los primeros 5 resultados y en vez de quedarse solo con un único resultado correcto, se van a considerar correctos todos los que sean adecuados para el producto. Cada uno de ellos se suma nuevamente ponderados por su posición como $\frac{1}{posición}$ obteniendo al final una puntuación para cada producto. Esto se hace con los 20 productos, se suma el resultado final y se obtiene la puntuación correspondiente a cada modelo. En este caso, la mejor puntuación posible es 45,7 aproximadamente.

$$\text{Métrica 2} = \sum_{i=1}^{20} \sum_{j=1}^{j_{max} \leq 5} \frac{1}{\text{posición de atributo correcto j para producto i}}$$

4.4.2. Clasificación de las distintas arquitecturas

Como se ha podido comprobar, los mejores resultados son los proporcionados por la intersección entre la búsqueda con BM25 y la búsqueda con SBERT separadamente, limpiando los productos previamente con `Spacy`. En general esta limpieza ha beneficiado a todas las arquitecturas implementadas. Aún así, sus valores están lejos de ser perfectos. Para la métrica 1 se obtienen 12 puntos de un máximo de 20, pero para la métrica 2 solo un 22,7 de 45,7. Esto indica que el modelo es mejor proporcionando un resultado bueno en la posición adecuada que proporcionando resultados buenos en general en todas sus posiciones. Como mucho, este modelo sirve para simplificar un trabajo que de otro modo tendría que ser puramente manual, pero no exime de la revisión de sus resultados por parte de un equipo humano.

Aún así, la herramienta es útil, y cuando se tienen del orden de ciento de miles de proyectos cualquier automatización es bienvenida. Por otro lado, la arquitectura para implementar estas herramientas hace del cambio de modelo algo trivial, con lo que posibles mejoras futuras en SBERT podrán ser actualizadas en el sistema de búsqueda fácilmente.

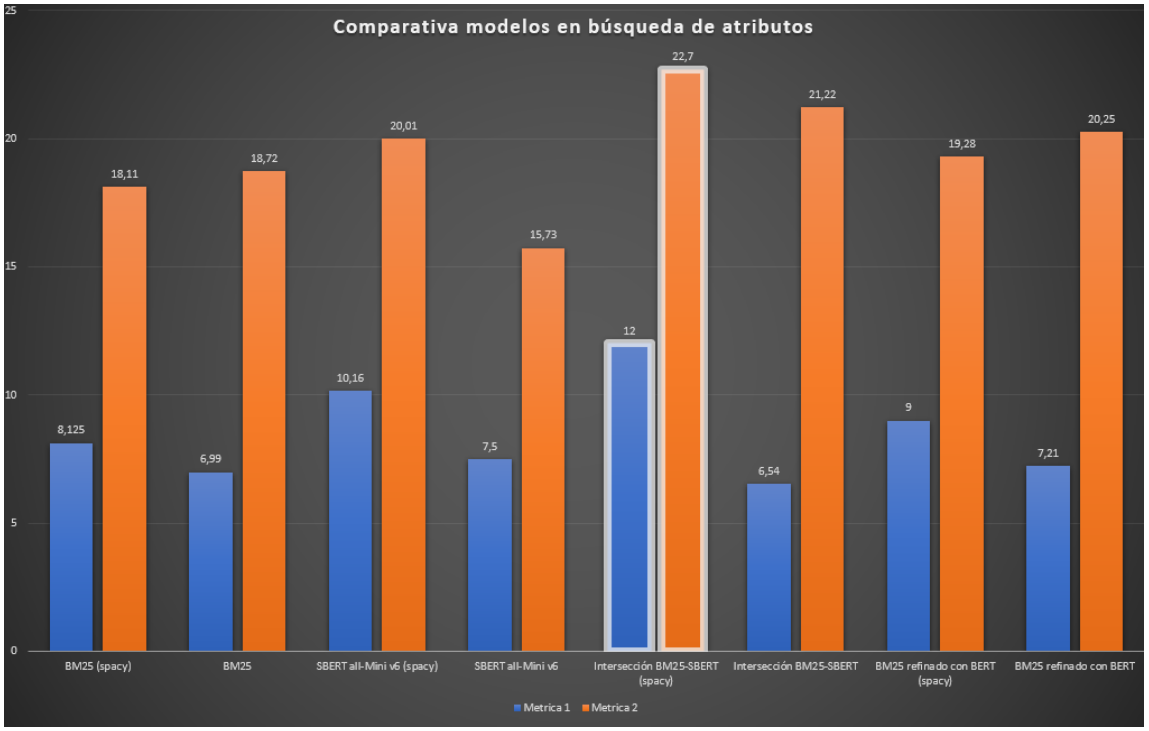

Figura 4.3: Comparativa de distintos modelos en la tarea de búsqueda y recuperación de información.

4.4.3. Mejoras

Como trabajo a futuro se propone investigar el modelo con arquitectura *transformer* entrenado en un *corpus* en español más grande hasta la fecha: MarIA [22]. El proyecto consta de dos modelos de estructura RoBERTa, y dos con estructura GPT-2, uno *base* y otro *large* en cada caso. Sería interesante comprobar los resultados de la búsqueda entrenando una red siamesa de tipo Sentence-BERT sobre uno de los modelos *encoder-only* de este proyecto.

Pero aunque esta pueda ser una posible vía prometedora para la mejora de resultados, no parece ser que sea la solución definitiva. Implementando esta metodología con modelos entrenados en *corpus* en inglés, y realizando la busqueda sobre productos en el mismo idioma, los resultado tampoco son perfectos. Esto parece estar relacionado con la propia construcción matemática de los *transformers*. En el próximo capítulo se va a presentar una posible vía de investigación para mejorar estos modelos del lenguaje desde una base teórica.

Capítulo 5

Y luego, ¿qué?

Los *tranformers* son el estado del arte actual en cuanto a NLP se refiere. Pero, como muestran los resultados del Capítulo 4, distan de comprender el lenguaje como lo haría un ser humano. En este capítulo se va a seguir la línea filosófica que defiende que para obtener una comprensión semántica humana, es necesario algo más que buscar correlaciones entre datos textuales.

5.1 De Chomsky a Judea Pearl

Varias veces en este trabajo se han señalado las teorías lingüísticas y cómo los *transformers* representan una de ellas en particular. También se ha mencionado a Chomsky en dos ocasiones: en primer lugar para señalar cómo hacía la distinción entre semántica (significados) y sintáctica (estructuras); y en segundo lugar para señalar cómo pensaba que las estructuras sintácticas (gramática) son un resultado inherente a nuestra biología.

Los esfuerzos de Chomsky se centraron en expresar de manera formal las reglas estructurales que fundamentan la gramática de los lenguajes naturales. Un *transformer*, mediante correlaciones, es capaz de simular estas reglas sintácticas, además de acercarse a establecer significados semánticos. Y sin embargo, no parece ser suficiente para reproducir de manera fidedigna el pensamiento humano. A diferencia de Chomsky, en este trabajo no se plantea que el "programa en nuestros genes" sea únicamente para generar sintaxis, sino que además es el causante de la flexibilidad semántica necesaria para imaginar. Esto es, para crear significados nuevos no extraídos de ninguna correlación vista en los datos de entrenamiento.

En su versión más extrema existen hipótesis (Sapir-Worf) que establecen que el lenguaje y el pensamiento se relacionan con una relación de equivalencia, sin orden jerárquico definido. Puede que esto sea así, y que simplemente en la descripción fundamental del lenguaje falte una pieza clave. De encontrarla, quizás un modelo entrenado con ella pudiera llegar a tener sentido común, y construir pensamientos entendidos como humanos. Puede que no sea así, y a pesar de establecer una descripción formal del funcionamiento de los lenguajes naturales no se llegue a simular el pensamiento humano en su totalidad.

En cualquier caso, encontrar una pieza teórica, probarla en el laboratorio informático actual, y mejorar nuestro entendimiento del funcionamiento de nuestro cerebro es un objetivo en sí mismo. Uno de los aspectos más importantes del estado actual de esta era de Aprendizaje Profundo e Inteligencia artificial es precisamente la posibilidad que genera a la hora de experimentar con teorías lingüísticas, filosóficas y matemáticas. Al fin y al cabo, las matemáticas son un estudio sobre las leyes y resultados de la inferencia cimentado sobre el lenguaje.

Chomsky era consciente de lo incompleto de sus teorías. En este trabajo se intenta justificar cómo la inferencia causal propuesta por Judea Pearl puede dar cuenta de algunas de sus carencias.

5.1.1. Causalidad

Uno de los debates más sonados en el mundo académico es quizás el dé una teoría física que de cuenta de todas las fuerzas fundamentales. Actualmente, la gravedad encuentra su mejor descripción bajo el marco geométrico de la Relatividad General, mientras que las demás fuerzas lo hacen bajo el de la Teoría Cuántica de Campos. Dos interpretaciones fundamentalmente distintas, cada una describiendo un aspecto distinto de nuestra percepción de la realidad.

El objetivo, por supuesto, es encontrar una teoría unificada: las famosas teorías de cuerdas, o la más reciente teoría discreta de Wolfram, entre otras, intentan dar cuenta de ello. Sin embargo, es importante señalar que, explicar un aspecto de la realidad con una interpretación, no anula a las demás para explicar el mismo. Dicho de otra forma, un aspecto de la realidad puede estar sujeto a varias descripciones. Simplemente puede haber alguna que sea más general, más fina, que explique un rango más amplio de aspectos y, por tanto, se considere como más adecuada para ser la descripción dominante.

El mensaje aquí es que no se presupone que ninguna de estas teorías "sea" la realidad, simplemente se consideran "descripciones" que dan cuenta de forma constructiva (esto es, utilizando las leyes de la inferencia demostradas con el lenguaje formal matemático) de resultados obtenidos experimentalmente, y que permiten predecir otros resultados que a su vez deberán coincidir con los experimentales para validarse. Se podría decir que son formas complejas de correlacionar datos, pero no son necesariamente únicas (varias pueden dar cuenta de los mismos resultados) y no aspiran a ser llamadas "realidad".

Lo que es fundamentalmente distinto de la inferencia causal, es que se construye sobre la máxima filosófica de que, aunque el mecanismo causal detrás del funcionamiento de un sistema a estudiar no sea observable, las trazas del mismo sí que lo son. Es decir, se asume que los datos generados y la "realidad" son fundamentalmente distintas, y se intenta hacer frente al problema de decir qué "es" la realidad. Distintos mecanismos causales tendrán como resultado distintos datos, por tanto la elección de uno u otro ya no está sujeta a interpretación. De esta máxima se derivan dos objetivos para cualquier marco de inferencia causal:

1. Los mecanismos causales de cualquier sistema a investigar tienen que tenerse en cuenta (de hecho tienen que formalizarse) en el análisis del mismo.

2. La colección de mecanismos (aunque la mayoría sean inobservables) deben estar formalmente ligados a su resultado: el fenómeno generado y los conjuntos de datos correspondientes.

Con respecto al primer objetivo, se estipula que la realidad puede ser representada de forma natural por un conjunto de mecanismos causales en la forma de un objeto matemático denominado "Modelo Causal Estructural", o *Structural Causal Model* (SCM). A menudo se representan como grafos acíclicos denominados grafos causales. En la práctica, en muchos casos estos mecanismos son imposibles de discernir en sistemas complejos. Aún así, se presupone que existen independientemente de nuestra habilidad para discernirlos.

Con respecto al segundo objetivo, Pearl se dio cuenta de que cada SCM inducía una jerarquía causal ("escalera de causalidad") que estratifica cualitativamente los distintos aspectos de la realidad subyacente. Consta de tres escalones, cada uno asociado a un concepto: la asociación, la intervención y el contrafáctico,

que responden a *grosso modo* con las actividades humanas de ver, hacer e imaginar, respectivamente (Figura 5.5).

Se va a ver otro ejemplo, este algo más concreto, para entender cómo contextualizar todo esto. Supóngase la siguiente situación: un prisionero de guerra que se enfrenta a su fusilamiento de la mano de dos soldados, dirigidos por su capitán de pelotón, que a su vez responde ante un tribunal que valora si se debe aplicar la pena máxima. Nótese que la crudeza del ejemplo ilustra perfectamente la dificultad de obtener datos del mismo. En estadística clásica, si se dispone de suficientes datos, se pueden fijar variables aleatorias para distintas acciones dentro de esta situación. Por ejemplo, se podrían asignar variables T, C, A, B, M que adopten valores en un espacio binario (sí/no) para las acciones de "tribunal dictamina ejecución", "capitán da orden de disparar", "soldado A dispara", "soldado B dispara" y "prisionero muere", respectivamente. Luego, con un número suficiente de sucesos se podrían calcular distintas probabilidades condicionadas que relacionen estas variables: $P(M \equiv \text{sí}|\ T \equiv \text{sí},\ C \equiv \text{sí})$, $P(M \equiv \text{sí}|\ T \equiv \text{sí},\ C \equiv \text{no}, A \equiv \text{sí})$, etc. Claro que si los soldados y el capitán son modelos en sus tareas, y respetan la cadena de mando, probabilidades como $P(M \equiv \text{sí}|\ T \equiv \text{sí},\ C \equiv \text{no}, A \equiv \text{no})$ serían nulas. De hecho, solo se tendrían dos sucesos que se repetirían constantemente: el que fija todas las variables en "si" y el que las fija en "no".

Ahora, supóngase que solo podemos *observar* lo que hace el soldado A, y se ve que dispara. ¿Cuál es la probabilidad de que el soldado B haya disparado? Es decir, ¿cuál es $P(B \equiv si|A \equiv si)$? La respuesta más evidente es que $P(B \equiv si|A \equiv si) = 1$, y es fácilmente calculable con estadística clásica. Sin embargo, ¿qué pasa si el soldado A decide saltarse la cadena de mando y disparar? En principio es un suceso que se corresponde con la situación planteada, pero si no forma parte de los sucesos recogidos para construir las probabilidades, no se tiene forma de utilizarlo en las predicciones.

Es aquí donde toma importancia la inferencia causal. Para empezar, distingue formalmente entre "observar" e "intervenir". No es lo mismo observar como el soldado A dispara que intervenir para que lo haga. La probabilidad de que, por ejemplo, el soldado B dispare si se observa que el soldado A ha disparado, es distinta a la probabilidad de que el soldado B dispare "obligando" al soldado A a disparar. Para distinguir esto, se expresa la intervención con el operador $do()$. En este ejemplo se estaría estipulando intuitivamente que $P(B|A)$ es fundamentalmente distinta a $P(B|do(A))$. $do(A)$ es un suceso que se sale de los observados, pero aún así es posible sacar información a partir de él de forma intuitiva. Por ejemplo, si su puntería no ha empeorado $P(M|do(A)) = 1$. ¡Y esta es una probabilidad que no se calcula a partir de los datos!

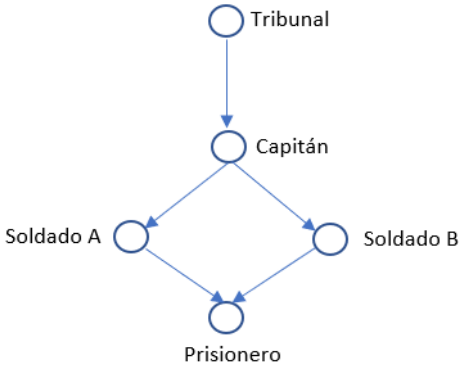

Figura 5.1: Ejemplo diagrama causal.

Según la inferencia causal, la pieza clave que le falta a la estadística para poder responder a preguntas tan relevantes en un sistema como este es precisamente darse cuenta de que detrás de los datos existe

un modelo causal estructural, y que intervenir en cualquier parte del mismo altera su estructura. En el ejemplo se tendría el diagrama casual de la Figura 5.1. Sin embargo, al preguntarse cuáles son las probabilidades asociadas al suceso $do(A)$, se alteraría en diagrama de la Figura 5.2.

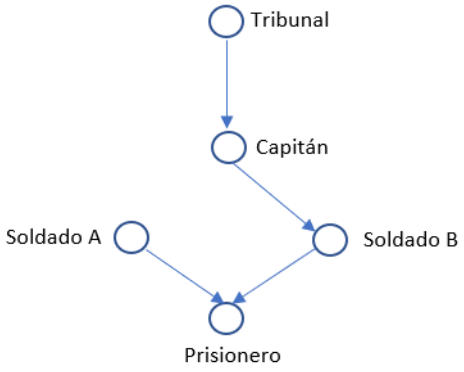

Figura 5.2: Ejemplo diagrama con *do(A=si)*

La escalera de causalidad

Como antes se mencionaba, se demuestra que, aceptando la existencia de diagramas causales como modelos de realidad [23], se deduce una jerarquización de niveles de razonamiento conocida como la escalera de causalidad, representada en la Figura 5.5. No se va a profundizar demasiado en ella por exceder los contenidos de este trabajo, pero en [5] se ofrece una explicación muy intuitiva al respecto.

Primer escalón: correlaciones. Es en este escalón, el nivel más bajo de razonamiento causal, donde se engloba al lenguaje de la probabilidad tradicional y, por ende, al Aprendizaje Automático actual. Si el lenguaje es una puerta a nuestros pensamientos, ya sea dándoles forma o expresándolos, entonces los *transformers* por sí solos son una herramienta inadecuada para su modelización según esta clasificación.

Segundo escalón: operador *do()*. Mientras que en el primer escalón se trataban observaciones pasivas, en este segundo escalón se va a lidiar con las intervenciones, y formalmente se va a utilizar el conocido como operador *do()*.

Tercer escalón: contrafácticos. En este escalón se intentan resolver las preguntas del tipo "¿qué pasaría si en vez de A hubiera pasado B?". Constituyen la forma más avanzada de razonamiento según Pearl.

5.2 Modelo de Razonamiento Contrafáctico (CRM)

En esta sección se va a ver una posible aplicación de los conceptos explicados de inferencia causal al mundo del procesamiento de lenguaje natural [24]. Asumiendo que el entendimiento humano se basa en establecer relaciones jerárquicas que pueden ser representadas por diagramas causales, el significado "oculto" del lenguaje no solo consiste en una relación probabilística entre sus elementos, sino que también consiste en un diagrama causal que los relaciona. Presumiblemente, la sintáctica de alguna forma induzca una estructura causal de base, pero el problema por supuesto es que, para cada situación y contexto del que se pueda hablar con un lenguaje natural, se podrán tener un número arbitrario de diagramas causales

basados en su semántica. Presumiblemente también, un acercamiento clásico de implementación de reglas *ad hoc* para la construcción de estos diagramas sea inviable. Es por eso que, en el acercamiento que se va a presentar en esta sección, se va a intentar plantear una arquitectura de red neuronal que aprenda a simular el efecto de tener diagramas causales que relacionen los elementos del lenguaje natural, utilizando para ello un aprendizaje supervisado sobre un conjunto de oraciones contrafácticas.

Considérese la oración "el matrimonio es la principal causa de divorcio" y la tarea de clasificar su intención en "neutro" o "irónica". Desde un punto de vista logístico, esta frase es perfectamente válida: es sintácticamente correcta y no parece haber contradicción en su significado. Estadísticamente, si representásemos en una tabla sujetos que se hayan divorciado, junto a sus características, incluyendo el hecho de que estén casados entre ellas, sería evidente la correlación entre divorcio y matrimonio. De manera análoga, un *transformer* es capaz de ver esta correlación en los textos con lo que se entrena, por tanto no sería descabellado suponer que clasificase la oración como "neutra".

Figura 5.3: Diagrama causal matrimonio ⟶ divorcio.

Pero el lenguaje natural es más complicado que eso y el lector hispano-hablante es capaz de detectar que la frase es "irónica". ¿Por qué? Una posible explicación es la siguiente: el diagrama causal que relaciona "matrimonio" y "divorcio" no es directo (Figura 5.3), sino que tiene un paso intermedio donde se encontrarían causas como "infidelidad", "celos", "ansiedad", etc (Figura 5.4). Seguramente se pueda encontrar un diagrama causal mejor, pero lo que se quiere argumentar es que la causa "matrimonio", pertenece a un estrato jerárquico "distinto" a las demás, y es esto lo que genera la connotación irónica.

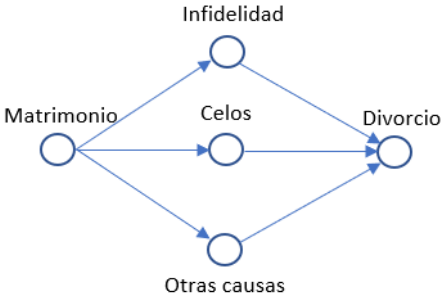

Figura 5.4: Diagrama causal matrimonio ⟶ causas directas ⟶ divorcio.

La clave para resolver este problema de clasificar correctamente el significado de una frase irónica es imitar el razonamiento contrafáctico humano. En esta sección se hace: 1) construyendo ejemplos contrafácticos para un ejemplo fáctico objetivo; 2) llamando a un modelo de lenguaje entrenado que haga predicciones para los ejemplos contrafácticos; y 3) comparando los ejemplos contrafácticos y fácticos para hacer una retrospección sobre la predicción del modelo [24]. Claro que este proceso no es nada trivial porque: 1) cualquier variante del ejemplo fáctico objetivo es válida para ser su contrafáctico, por tanto el espacio de los mismos es enorme; y 2) el mecanismo según el cual hacemos la retrospección todavía no está claro, con lo que reproducirlo formalmente es notablemente difícil.

En aras de este objetivo, en el artículo [24] se propone un modelo de razonamiento contrafáctico que consta de dos módulos: el módulo generativo y el módulo de retrospección. En particular, dado un ejemplo fáctico en la fase de *test*, el módulo generativo construye ejemplos contrafácticos representativos

imaginándose "cuál sería el contenido si la etiqueta del ejemplo fuese y". Para imitar el mecanismo desconocido de retrospección humana, se construye un módulo de retrospección con una red neuronal profunda cuidadosamente diseñada que compara separadamente la representación latente y la predicción (la intención según tres etiquetas: positiva, negativa y neutra) de los ejemplos fácticos y contrafácticos.

5.2.1. Formulación del problema

Datos de entrenamiento:

- Ejemplos fácticos: $\mathcal{T} = \{(\boldsymbol{x}, y)\}$ donde $y \in [C]$ denota la clase o la etiqueta objetivo, $\boldsymbol{x} \in \mathbb{R}^D$ es la representación realizada por el modelo de lenguaje (BERT, RoBERTa, etcétera).
- Ejemplos contrafácticos: $\mathcal{T}^* = \{(\boldsymbol{x}_c^*, c) \quad | \quad (\boldsymbol{x}, y) \in \mathcal{T}, c \in [C], c \neq y\}$ donde $(\boldsymbol{x}_c^*, c)$ es el ejemplo contrafáctico en la clase c correspondiente al ejemplo fáctico $(\boldsymbol{x}, y)$. Dados los ejemplos fácticos etiquetados, los contrafácticos pueden construirse, bien manualmente o bien automáticamente, realizando cambios mínimos en $\boldsymbol{x}$ para cambiar su etiqueta de y a c.

Se asume que BERT se afina sobre estos datos. Formalmente:

$$\hat{\boldsymbol{\theta}} = \min_{\theta} \sum_{(\boldsymbol{x},y)\in\mathcal{T}/\mathcal{T}^*} l(y, f(\boldsymbol{x}, \boldsymbol{\theta})) + \alpha||\boldsymbol{\theta}||, \tag{5.1}$$

donde $\hat{\boldsymbol{\theta}}$ son los parámetros aprendidos del modelo $f(\cdot)$, $l(\cdot)$ es una función de coste para clasificación, como la de entropía-cruzada[25], y α es un hiper-parámetro para ajustar la regularización.

El objetivo es construir un proceso de toma de decisiones que simule el razonamiento contrafáctico una vez entrenado. Dado un ejemplo $\boldsymbol{x}$ del grupo de *test*, se trata de clasificarlo correctamente en cuanto a las clases que se consideren, generando ejemplos contrafácticos y retrospeccionando la decisión, lo cual se formula como:

$$\boldsymbol{y} = h\left(\boldsymbol{x}, \{\boldsymbol{x}^*\}|\boldsymbol{\eta}, \hat{\boldsymbol{\theta}}\right), \{\boldsymbol{x^*}\} = g(\boldsymbol{x}|\boldsymbol{\omega}),$$

$\boldsymbol{y} \in \mathbb{R}^C$ denota la predicción final para el ejemplo de *test* $\boldsymbol{x}$, que es una distribución sobre las distintas etiquetas de clase; $\boldsymbol{x^*}$ es uno de los ejemplos contrafácticos generados para $\boldsymbol{x}$. Se espera que el módulo generativo $g(\cdot)$ parametrizado por $\boldsymbol{\omega}$ construya un conjunto adecuado de ejemplos contrafácticos para el ejemplo fáctico correspondiente, lo cual dota de señales al módulo de retrospección $h(\cdot)$ parametrizado por $\boldsymbol{\eta}$ para retrospeccionar la predicción $f(\boldsymbol{x}, \hat{\boldsymbol{\theta}})$ dada por el modelo de BERT (adaptado a la clasificación) ya entrenado. En particular, $h(\cdot)$ y $g(\cdot)$ se aprenderán de los ejemplos de entrenamiento fácticos y contrafácticos, respectivamente.

5.2.2. Módulo de retrospección

Comparación de las representaciones

Dada una pareja de ejemplos $\boldsymbol{x}$ y $\boldsymbol{x^*}$, fáctico y contrafáctico, respectivamente, se considera que la información práctica para el aprendizaje se encuentra en la diferencia entre los mismos. Es por eso que en este bloque se calcula $\boldsymbol{y}_\Delta = f(\boldsymbol{x} - \boldsymbol{x^*}|\hat{\boldsymbol{\theta}})$ donde $\boldsymbol{y}_\Delta \in \mathbb{R}^C$ denota la predicción de la diferencia $\boldsymbol{x} - \boldsymbol{x^*}$ dada por BERT adaptado a la clasificiación. Es importante señalar que se utiliza un duplicado de este modelo de clasificación de manera que el entrenamiento de este modulo de retrospección no afecte a los parámetros del afinado de BERT.

Comparación de las predicciones

Para hacer retrospección en la predicción $f(\boldsymbol{x}|\hat{\boldsymbol{\theta}})$, este bloque va a comparar las predicciones dadas por cada pareja de fáctico-contrafáctico. La idea es buscar patrones entre las predicciones $f(\boldsymbol{x}|\hat{\boldsymbol{\theta}})$, $f(\boldsymbol{x}^*|\hat{\boldsymbol{\theta}})$ y $\boldsymbol{y}_\Delta$, implementando para ello una red convolucional (CNN) y hacer una nueva predicción $\boldsymbol{y}^*$ a partir de la misma:

$$\boldsymbol{y}^* = \text{CNN}\left(f(\boldsymbol{x}, |\hat{\boldsymbol{\theta}}), f(\boldsymbol{x}^*, |\hat{\boldsymbol{\theta}}), \boldsymbol{y}_\Delta\right) \tag{5.2}$$

En particular una capa *stack* "apila" las tres predicciones en una matriz, que hace las veces de "imagen" para facilitar la búsqueda de patrones. Formalmente $\boldsymbol{Y} = [f(\boldsymbol{x}, |\hat{\boldsymbol{\theta}}), f(\boldsymbol{x}^*, |\hat{\boldsymbol{\theta}}), \boldsymbol{y}_\Delta]$ donde $\boldsymbol{Y} \in \mathbb{R}^{C\times 3}$. Una vez creada, $\boldsymbol{Y}$ es introducida a una red neuronal convolucional de una dimensión:

$$\boldsymbol{H} = \sigma(\boldsymbol{Y}\boldsymbol{F}), \quad \boldsymbol{H_{ij}} = \sigma(\boldsymbol{Y}[:, i]\boldsymbol{F}[j, :]) \tag{5.3}$$

donde $\boldsymbol{F} \in \mathbb{R}^{3\times K}$ denota los filtros de la red convolucional y $\sigma(\cdot)$ es una función de activación (por ejemplo GELU). $\boldsymbol{Y}[:, i]$ representa la i-ésima fila de $\boldsymbol{Y}$ y $\boldsymbol{F}[j, :]$ la j-ésima columna de $\boldsymbol{F}$. El filtro $\boldsymbol{F}[j, :]$ puede aprender reglas para realizar la retrospección. Por ejemplo, un filtro de valores $[1, -1, 0]$ deduce la predicción del ejemplo contrafáctico a partir del ejemplo fáctico. El *output* $\boldsymbol{H} \in \mathbb{R}^{C\times K}$ se "aplana" (*flatten*) en un vector llamado $\boldsymbol{B}_{flatten}$ y se introduce en una capa de red neuronal densa (*fully-connected layer*, FC) para capturar los patrones de cada clase. Formalmente:

$$\boldsymbol{y}^* = \boldsymbol{W}\boldsymbol{B}_{flatten} + \boldsymbol{b}, \tag{5.4}$$

donde $\boldsymbol{W}$ y $\boldsymbol{b}$ son los parámetros de la capa FC.

Fusión

El objetivo es fusionar las predicciones obtenidas al realizar la retrospección $\{\boldsymbol{y}^*\}$ en una decisión final $\boldsymbol{y}$. La forma de hacerlo en el artículo [24] es utilizando una capa de *pooling*, de manera que $\boldsymbol{y} = \text{pooling}(\{\boldsymbol{y}^*\})$.

Como esta fusión está hecha después del bloque de predicción, se la conoce como *late fusion*. Existen otras versiones de este módulo donde la fusión se realiza en la CNN durante la comparación (*middle fusion*) o previamente a la CNN (*early fusion*). Lo cierto es que no está claro cómo funciona el mecanismo de retrospección en nuestros cerebros, de manera que cualquiera de las tres arquitecturas es válida.

Entrenamiento

Los parámetros del módulo de retrospección se actualizan según la función de perdida de clasificación sobre los datos de entrenamiento:

$$\hat{\boldsymbol{\eta}} = \min_{\eta} \sum_{(\boldsymbol{x},y)\in\mathcal{T}} l(y, \boldsymbol{y}) + \lambda||\boldsymbol{\eta}|| \tag{5.5}$$

donde λ denota los hiperparámetros para ajustar el término de regularización.

5.2.3. Módulo de generación

El objetivo de este módulo es construir ejemplos contrafácticos que sean útiles a la hora de realizar la retrospección. Como la tarea consiste en tomar decisiones sobre C etiquetas de clase, los autores se

hacen la pregunta de "¿cuál sería el contenido si el ejemplo perteneciera a la clase $c \in [C]$?", generando así C ejemplos contrafácticos $\{\boldsymbol{x}_c^*\}_{c\in[C]}$. Con las C clases como objetivos, el espacio de contrafácticos posibles se reduce drásticamente. El módulo de generación se divide en: 1) un bloque de descomposición del ejemplo fáctico $\boldsymbol{x}$ para limpiar el contenido irrelevante para la etiqueta de la clase, y que da como resultado $\boldsymbol{u} = d(\boldsymbol{x}|\boldsymbol{\omega})$; 2) inyección de la clase c en $\boldsymbol{u}$ para generar el ejemplo contrafáctico $\boldsymbol{x}_c^*$.

Descomposición

Para "destilar" $\boldsymbol{u}$ es necesario encontrar la conexión entre el contenido del ejemplo fáctico y cada clase. Por tanto, se tienen en cuenta las representaciones de las clases en la función de descomposición. Para alinear el espacio de muestreo del módulo de generación con el modulo de retrospección $h(\cdot)$ y con el modelo de clasificación $f(\cdot)$, se extraen los parámetros de la capa de predicción del modelo de clasificación y se toman como las representaciones de cada clase. En particular, se extrae la matriz $\boldsymbol{W} \in \mathbb{R}^{C\times D}$ donde la c-ésima fila se corresponde con la clase c. Nótese que se asume que la capa de predicción tiene la misma dimensión que las representaciones latentes dadas por BERT (lo cual, actualmente, es un ajuste común en la mayoría de modelos de lenguaje de este tipo). La función de descomposición se diseña utilizando una red convolucional CNN con el objetivo citado: averiguar las correlaciones entre cada ejemplo fáctico y las distintas clases. El proceso es análogo al utilizado en el bloque de predicción del módulo de retrospección:

1. Capa *stack*: En primer lugar, se "apilan" las representaciones del ejemplo fáctico y de las clases, y el producto elemento a elemento $\odot$ entre cada ejemplo y cada clase: $\boldsymbol{X} = [\boldsymbol{x}, \boldsymbol{W}^T, \boldsymbol{x} \odot \boldsymbol{W}^T]$. La idea es que el producto elemento a elemento $\boldsymbol{x} \odot \boldsymbol{W}^T \in \mathbb{R}^{D\times C}$ aporte información sobre el parecido de cada dimensión de $\boldsymbol{x}$ con cada clase: grandes valores absolutos indican una conexión fuerte.

2. Capa *convolucional*: Al igual que en predicción, se elige una capa con filtros unidimensionales, pero en este caso lo que se intenta es que detecten contenidos del ejemplo relevante para su clasificación en cada clase. Formalmente, en esta capa se calcula $\boldsymbol{h} = pooling(\sigma(\boldsymbol{X}\boldsymbol{F}^g))$, donde $\boldsymbol{F}^g \in \mathbb{R}^{(2C+1)\times L}$ denota los L filtros de la red. El *output* $\boldsymbol{h} \in \mathbb{R}^D$ es una representación oculta.

3. Capas FC: Se utilizan dos capas de red neuronal lineal FC, formalmente: $\boldsymbol{u} = \boldsymbol{W_2}\sigma(\boldsymbol{W_1}\boldsymbol{h} + \boldsymbol{b_1}) + \boldsymbol{b_2}$, donde $\boldsymbol{W_1} \in \mathbb{R}^{M\times D}$, $\boldsymbol{W_2} \in \mathbb{R}^{D\times M}$, $\boldsymbol{b_2} \in \mathbb{R}^D$, y $\boldsymbol{b_1} \in \mathbb{R}^M$ son parámetros de aprendizaje. M es un hiper-parámetro para ajustar la complejidad de la función de descomposición. Nótese que se pueden "apilar" más capas para aproximar mejor la función [12].

   Los parámetros de la función de descomposición se aprenden optimizando:

$$\min_{\omega} \sum_{(\boldsymbol{x}_c^*, c)\in\mathcal{T}^*} r(\boldsymbol{u}_c^*, \tilde{\boldsymbol{u}}_c) + \gamma l(c, f(\boldsymbol{x}_c^* - \boldsymbol{u}_c^*|\hat{\boldsymbol{\theta}}))$$

$$+ r(\boldsymbol{u}, \tilde{\boldsymbol{u}}_c) + \gamma l(y, f(\boldsymbol{x} - \boldsymbol{u}|\hat{\boldsymbol{\theta}})), \tag{5.6}$$

   donde $\boldsymbol{u}_c^* = d(\boldsymbol{x}_c^*|\boldsymbol{\omega})$ y $\boldsymbol{u} = d(\boldsymbol{x}|\boldsymbol{\omega})$ son los resultados de la descomposición del ejemplo contrafáctico $\boldsymbol{x}_c^*$ y del ejemplo fáctico correspondiente $\boldsymbol{x}$; $\tilde{\boldsymbol{u}}_c = \frac{1}{2}(\boldsymbol{x} + \boldsymbol{x}_c^*)$ denota el valor objetivo de la descomposición. Por su parte $r(\cdot)$ y $l(\cdot)$ son la distancia Euclídea y la función de pérdida de clasificación. Minimizando ambos términos, se busca que el resultado de la descomposición: 1) sea próximo al valor objetivo; 2) si se deduce del ejemplo original (e.g., $\boldsymbol{x} - \boldsymbol{u}$) la clasificación no se vea afectada. γ es un hiper-parámetro para equilibrar ambos términos.

   Se fija que el contenido irrelevante para las clases de $\boldsymbol{x}$ y $\boldsymbol{x}_c^*$ sea $\tilde{\boldsymbol{u}}_c = \frac{1}{2}(\boldsymbol{x} + \boldsymbol{x}_c^*)$ basándose en la regla del paralelogramo. Nótese que este par de ejemplos pertenecen a dos clases y y c distintas,

separadas por un límite o hiperplano. Considerando que $\boldsymbol{x}$ se corresponde con un vector en el espacio oculto, se puede descomponer el mismo en dos componentes que son ortogonales y paralelas a dicho hiperplano: $\boldsymbol{x}_c^* = \boldsymbol{o}_c^* + \boldsymbol{p}_c^*$ y $\boldsymbol{x} = \boldsymbol{o} + \boldsymbol{p}$. Puesto que los dos ejemplos pertenecen a clases distintas, sus componentes ortogonales están en direcciones opuestas y su suma solo contendrá las componentes paralelas, que son irrelevantes a la hora de decidir entre las clases y y c.

Inyección

Análogamente, dado un ejemplo de prueba $\boldsymbol{x}$, podemos inyectar las componentes ortogonales hacia la clase c: $\boldsymbol{x}_c^* = 2 * d(\boldsymbol{x}|\hat{\boldsymbol{\omega}}_c) - \boldsymbol{x}$, que es el contenido imaginando que el ejemplo pertenece a la clase c. Cada ejemplo de prueba se inyecta sobre todas las clases, obteniendo C ejemplos contrafácticos $\{\boldsymbol{x}_c^*\}$, que son después utilizados en el módulo de retrospección.

5.2.4. Paradigma de aprendizaje

El paradigma actual en la implementación de BERT para el análisis de sentimientos es hacer un ajuste fine sobre un conjunto de datos etiquetados. El CRM constituye un nuevo paradigma de aprendizaje, que aparece reflejado en el siguiente pseudocódigo.

Algorítmo 12: Paradigma de aprendizaje con CRM

Input: Datos de entrenamiento $\mathcal{T}, \mathcal{T}^*$
/* Entrenamiento del CRM.
1 Optimizar ecuación 5.1: **Entrenamiento del modelo de clasificación** a partir de BERT pre-entrenado.
2 Optimizar ecuación 5.6: **Entrenamiento del módulo de generación**.
3 Optimizar ecuación 5.5: **Entrenamiento del módulo de retrospección**.
4 return $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}_c$ y $\hat{\boldsymbol{\eta}}$.
/* Prueba del CRM.
5 Calcular $f((\boldsymbol{x}|\hat{\boldsymbol{\theta}}))$: **Inferencia con el modelo de clasificación.**
6 for $c \in C$ **do**:
7 $\quad \boldsymbol{x}_c^* = 2 * g(\boldsymbol{x}|\hat{\boldsymbol{\omega}}_c) - \boldsymbol{x}$; **Generación de contrafácticos**
8 end
9 Calcular $h(\boldsymbol{x}, \{\boldsymbol{x}_c^*\}|\hat{\boldsymbol{\eta}}, \hat{\boldsymbol{\theta}})$: **Retrospección**

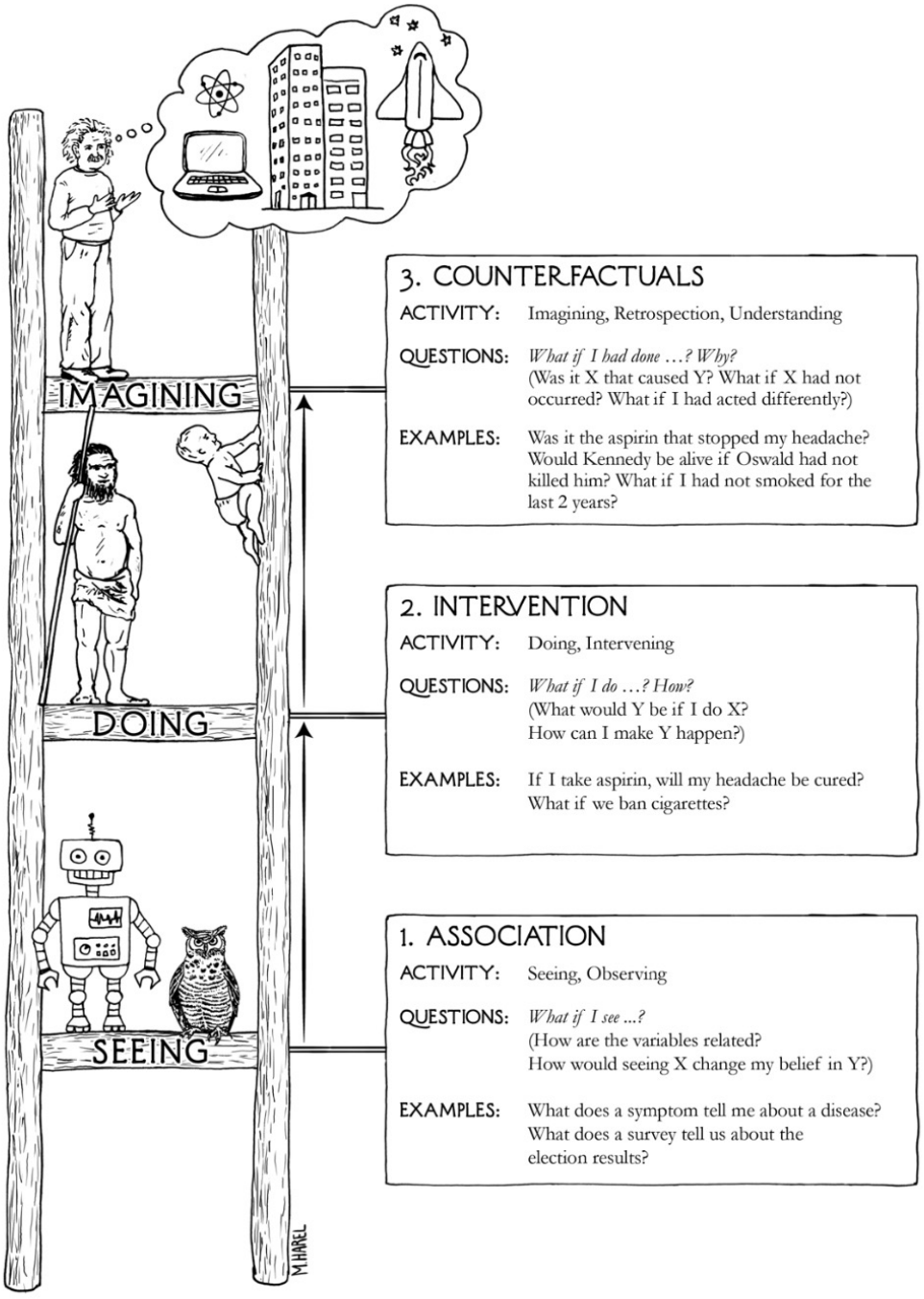

FIGURE 1.2. The Ladder of Causation, with representative organisms at each level. Most animals, as well as present-day learning machines, are on the first rung, learning from association. Tool users, such as early humans, are on the second rung if they act by planning and not merely by imitation. We can also use experiments to learn the effects of interventions, and presumably this is how babies acquire much of their causal knowledge. Counterfactual learners, on the top rung, can imagine worlds that do not exist and infer reasons for observed phenomena. (*Source:* Drawing by Maayan Harel.)

Figura 5.5: Representación de la escalera de causalidad extraída de *The book of why* [5].

Capítulo 6

Conclusiones

6.1 Conclusiones

En la actualidad el Aprendizaje Automático, y en particular el Aprendizaje Automático Profundo, es una herramienta de indudable éxito en numerosas tareas de distinta naturaleza. En este trabajo se ha explorado su aplicación al campo del Procesamiento del Lenguaje Natural, llegando a una serie de conclusiones:

- El *modus operandi* de utilizar algoritmos de *Deep Learning* en este campo, se corresponde con la teoría linguística ambientalista. Esto es: deja de lado las teorías constructivistas, como la de Chomsky, adoptando una aproximación puramente estadística para el procesado del lenguaje.

- Como se puede ver en la tarea de búsqueda y recuperación de información, la comprensión semántica de estos modelos del lenguaje dista mucho de ser óptima. Cuando el lenguaje se ve alterado, por ejemplo con nombres de artículos incompletos o sin una sintaxis definida, el desempeño de SBERT se encuentra por debajo, en algunos casos, de las técnicas tradicionales.

- El desempeño en un test de Turing parece ser mucho mayor con los modelos entrenados con *transformers* que nada que se haya alcanzado con un procedimiento simbólico. Esto parece indicar que no es un mal camino para alcanzar "máquinas que hablen". Por otro lado, a pesar de haber sido entrenadas con más información de la que ve cualquier ser humano siguen careciendo de una verdadera comprensión semántica. Es por eso que, sin renunciar por completo a esta dinámica ambientalista, se ha propuesto un complemento lógico que se desmarca de la estadística tradicional: la inferencia causal.

- Se presenta un estudio [24] en el que se implementan las ideas de la inferencia causal en Procesamiento del Lenguaje Natural mediante un modelo de razonamiento contrafáctico. En él, después de aplicárselo a distintas tareas típicas (análisis de sentimientos e inferencia de lenguaje natural), se consiguen mejoras del desempeño de modelos *encoder-only* (BERT, RoBERTa) (entre un 5 a 15,6 % según tarea y modelo).

6.2 Valoración del trabajo, cabos sueltos y posibles vías de investigación

Se podría decir que este trabajo ha constado de tres grande bloques:

1. Parte de este trabajo se ha centrado en la exploración y compresión del Procesamiento del Lenguaje Natural, y en particular de las técnicas de Aprendizaje Automático Profundo utilizadas en este campo. Concretamente, la arquitectura *transformer* ha sido un desafío en sí mismo, en gran medida por la falta de formalismo en la bibliografía más habitual.

2. En segundo lugar y desde un punto de vista técnico, la implementación en Python de estas tecnologías, en un entorno de producción real, y con un proyecto de *Big Data* que resultase escalable y operativo ha llevado más tiempo del que vale la pena mencionar en un trabajo de matemáticas.

3. El último gran bloque ha consistido en la comprensión desde el punto de vista lingüístico de las carencias de los grandes modelos del lenguaje, unido a la búsqueda de un planteamiento filosófico que pudiera ayudar a dar cuenta de ellas. El razonamiento causal y su formalización, la inferencia causal, representan un campo de las matemáticas en si mismas, así que es necesario un estudio previo para justificar su implementación en un campo particular como el del NLP.

Como es normal en un trabajo de estas dimensiones, han quedado algunos cabos sueltos:

1. Los vectores *query, key* y *value* utilizados en el mecanismo de atención. Su justificación filosófica no está clara. La explicación más habitual es que toman sus nombres del mundo de la búsqueda y recuperación de información, pero no sé relacionarlo con los conceptos de secuencia primaria y de contexto.

2. El mecanismo de retrospección utilizado en el CRM. En este caso, hasta los propios autores admiten que es un área de investigación aún abierta.

3. Aunque Chomsky hace la distinción entre sintáctica y semántica, el significado del lenguaje puede interpretarse como un resultado de su sintáctica. Y si las reglas de la sintaxis se interpretan como diagramas causales, quizás sería posible aplicar las leyes de la inferencia causal para realizar predicciones probabilísticas sobre las palabras dentro de una estructura sintáctica. Esta idea no está nada clara, pero merece la pena reflexionar sobre ella.

4. Desde el punto de vista de la complejidad computacional, sería interesante hacer un estudio de los algoritmos citados.

5. La aplicación de redes neuronales con construcción Bayesiana tampoco se ha explorado.

Bibliografía

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," 2017.

[2] N. Mambrol, "Transformational Generative Grammar," *Literary Theory and Criticism*, 2020.

[3] N. Chomsky, *Syntactic Structures*. De Gruyter Mouton, 2020.

[4] S. Carroll and G. Marcus, "Gary Marcus on Artificial Intelligence and Common Sense," 2022. https://www.preposterousuniverse.com/podcast/. Fecha de consulta: 15/08/22.

[5] J. Pearl and D. Mackenzie, *The Book of Why*. Basic Books, 2018.

[6] A. Ng, "CS229 Lecture notes - Supervised learning," *Stanford University*, 2012.

[7] A. Soleimany, "MIT 6.S191: Deep Learning New Frontiers," 2022. https://www.youtube.com/watch?v=wySXLRTxAGQ. Fecha de consulta: 25/08/22.

[8] M. Ali, *PyCaret: An open source, low-code machine learning library in Python*, April 2020.

[9] R. S. Olson and J. H. Moore, "TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning," in *Proceedings of the Workshop on Automatic Machine Learning* (F. Hutter, L. Kotthoff, and J. Vanschoren, eds.), vol. 64 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 66–74, PMLR, 2016.

[10] M. A. Nielsen, "The Universal Approximation Theorem for Neural Networks," *Determination Press*, 2015. Fecha de consulta: 25/07/22.

[11] 3Blue1Brown, "Neural Networks," https://www.3blue1brown.com/topics/neural-networks. Fecha de consulta: 12/05/22.

[12] M. A. Nielsen, "Neural Networks and Deep Learning," 2017. https://www.youtube.com/watch?v=Ijqkc7OLenI. Fecha de consulta: 24/07/22.

[13] C. Olah, "Neural Networks, Manifolds, and Topology," 2014. http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/. Fecha de consulta: 02/08/22.

[14] M. Phuong and M. Hutter, "Formal Algorithms for Transformers," 2022.

[15] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining approach," 2019.

[16] I. Kant, *Critique of Pure Reason*. Cambridge University Press, 1998.

[17] N. Reimers and I. Gurevych, *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019.

[18] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014.

[19] *Hugging Face*. https://huggingface.co. Fecha de consulta: 07/07/22.

[20] M. Honnibal and I. Montani, "spaCy: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing." 2017.

[21] *Google Cloud. Dataflow*. https://cloud.google.com/dataflow?hl=es. Fecha de consulta: 07/07/22.

[22] A. Gutiérrez-Fandiño, J. Armengol-Estapé, M. Pàmies, J. Llop-Palao, J. Silveira-Ocampo, C. P. Carrino, C. Armentano-Oller, C. Rodriguez-Penagos, A. Gonzalez-Agirre, and M. Villegas, "Maria: Spanish language models," *Procesamiento del Lenguaje Natural*, vol. 68, 2022.

[23] E. Bareinboim, J. D. Correa, D. Ibeling, and T. F. Icard, "On Pearl's Hierarchy and the Foundations of Causal Inference," *Probabilistic and Causal Inference*, 2022.

[24] F. Feng, J. Zhang, X. He, H. Zhang, and T.-S. Chua, "Empowering Language Understanding with Counterfactual Reasoning," vol. abs/2106.03046, 2021.

[25] Wikipedia, "Divergencia de Kullback-Leibler - Wikipedia, la enciclopedia libre," https://es.wikipedia.org/w/index.php?title=Divergencia_de_Kullback-Leibler&oldid=143037170. Fecha de consulta: 23/08/22.

Apéndice A

Lista de notación

Símbolo	Tipo	Explicación
$[N]$	$:=\{1, ..., N\}$	un conjunto de enteros 1, 2, ..., N-1, N
i,j	$\in \mathbb{N}$	índices genéricos de enteros
V	$\{x_1, ..., x_{N_V}\}$	vocabulario
N_V	$\in \mathbb{N}$	tamaño vocabulario
V^*	$= \cup_{\ell=0}^{\infty} V^\ell$	clausura de V
ℓ_{max}	$\in \mathbb{N}$	longitud máxima de secuencia
ℓ	$\in [\ell_{max}]$	longitud secuencia de tókenes
t	$\in [\ell]$	índice del *token* en la secuencia
$d_{...}$	$\in \mathbb{N}$	dimensión de varios vectores
$\boldsymbol{x}$	$\in V^*$	$= x_1...x_l$ secuencia de *tókenes* primaria
$\boldsymbol{z}$	$\in V^*$	$= z_1...z_l$ secuencia de *tókenes* de contexto
$M[i,j]$	$\in \mathbb{R}$	entrada M_{ij} de la matriz $M \in \mathbb{R}^{d \times d'}$
$M[i,:] \equiv M[i]$	$\in \mathbb{R}^{d'}$	*i-ésima* fila de la matriz $M \in \mathbb{R}^{d \times d'}$
$M[:,j]$	$\in \mathbb{R}^{d}$	*j-ésima* columna de matriz $M \in \mathbb{R}^{d \times d'}$
$\boldsymbol{e}$	$\in \mathbb{R}^{d_e}$	*input embedding* de un token
$\boldsymbol{X}$	$\in \mathbb{R}^{d_e \times \ell_x}$	matriz con los *input embeddings* de cada token de la secuencia primaria
$\boldsymbol{Z}$	$\in \mathbb{R}^{d_e \times \ell_z}$	matriz con los *input embeddings* de cada token de la secuencia de contexto
$Mask$	$\in \mathbb{R}^{\ell_z \times \ell_x}$	matriz *máscara*, determina el contexto de atención para cada *token*
L, L_{enc}, L_{dec}	$\in \mathbb{N}$	número de capas (*encode, decoder*)
l	$\in [L]$	índice de capa de la red
H	$\in \mathbb{N}$	número de *heads* de atención
h	$\in [H]$	índice de *head* de atención
N_{data}	$\in \mathbb{N}$	(i.i.d.) tamaño del conjunto de datos/secuencias
n	$\in [N_{data}]$	índice de la secuencia en el conjunto de datos
η	$\in (0, \infty)$	tasa de aprendizaje
τ	$\in (0, \infty)$	temperatura; controla el intercambio diversidad-plausibilidad durante la inferencia
$\boldsymbol{W_e}$	$\in \mathbb{R}^{d_e \times N_V}$	matriz *token embedding*
$\boldsymbol{W_p}$	$\in \mathbb{R}^{d_e \times \ell_{max}}$	matriz *positional embedding*
$\boldsymbol{W_u}$	$\in \mathbb{R}^{N_v \times d_e}$	matriz *unembedding*
$\boldsymbol{W_q}$	$\in \mathbb{R}^{d_{attn} \times d_x}$	matriz de pesos *query*
$\boldsymbol{b_q}$	$\in \mathbb{R}^{d_{attn}}$	sesgo *query*
$\boldsymbol{W_k}$	$\in \mathbb{R}^{d_{attn} \times d_z}$	matriz de pesos *key*
$\boldsymbol{b_k}$	$\in \mathbb{R}^{d_{attn}}$	sesgo *key*
$\boldsymbol{W_v}$	$\in \mathbb{R}^{d_{out} \times d_z}$	matriz de pesos *value*
$\boldsymbol{b_v}$	$\in \mathbb{R}^{d_{out}}$	sesgo *value*
$\boldsymbol{\mathcal{W}_{qkv}}$		colección de los parámetros de arriba de una capa de atención con una única *head*
$\boldsymbol{W_0}$	$\in \mathbb{R}^{d_{out} \times H d_{mid}}$	matriz de pesos de salida
$\boldsymbol{b_0}$	$\in \mathbb{R}^{d_{out}}$	sesgo de salida
$\boldsymbol{\mathcal{W}}$		colección de los parámetros superiores para una capa de atención *multi-head*
$\boldsymbol{W_{ffn}}$	$\in \mathbb{R}^{d_1 \times d_2}$	matriz de pesos correspondiente a una capa lineal en la FFN en un *transformer*
$\boldsymbol{b_{ffn}}$	$\in \mathbb{R}^{d_1}$	*sesgo* correspondiente a una capa lineal en la FFN en un *transformer*
$\boldsymbol{\gamma}$	$\in \mathbb{R}^{d_e}$	parámetro de tasa de aprendizaje en capa de normalización
$\boldsymbol{\beta}$	$\in \mathbb{R}^{d_e}$	parámetro *offset* de capa de normalización
$\boldsymbol{\theta}$,$\hat{\boldsymbol{\theta}}$	$\in \mathbb{R}^{d}$	colección de todos los parámetros de aprendizaje/aprendidos en un *Transformer*

Apéndice B

Probabilidad Bayesiana

B.1 El eterno debate: probabilidad bayesiana vs probabilidad frecuentista

Como se ha mencionado en la introducción, las matemáticas, de alguna forma, consisten en una forma altamente refinada de estudiar y expresar las leyes de la inferencia, sean estas universales o sujetas a nuestra naturaleza biológica. Por ende, no es de extrañar que la mejor forma de entender estas dos maneras de hacer estadística venga de un estudio en psicología.

En 1974, los psicólogos Amos Tversky y Daniel Kahneman publicaron un artículo en la revista Science de título *Juicio bajo incertidumbre: heurística y prejuicios*, por el cual Kanheman fue galardonado con un Premio Nobel en 2002. En él, aparece un ejemplo tremendamente representativo para el tema que se pretende abordar:

Supóngase un hombre de mediana edad llamado Jaime. Jaime es tímido e introvertido. También es útil y capaz, pero sin mucho interés en la gente o en el mundo real. Con una personalidad mansa y ordenada, tiene una necesidad de orden y estructura en su vida, y siente pasión por los detalles. Sabiendo esto, ¿qué es más probable: que Jaime sea bibliotecario o que sea granjero?

El grueso de los sujetos a los que se les presentó el problema respondieron que Jaime era bibliotecario. Después de todo, su personalidad iba más a juego con el estereotipo para este trabajo. Para Tversky y Kanhneman este hecho suponía un rasgo de irracionalidad en el pensamiento humano convencional, aunque hay debate sobre esta interpretación. Lo que sí que era evidente es que muchas de las respuestas parecían obviar por completo la proporción de granjeros frente a bibliotecarios en su región y época (unos 20:1 en EEUU durante esos años).

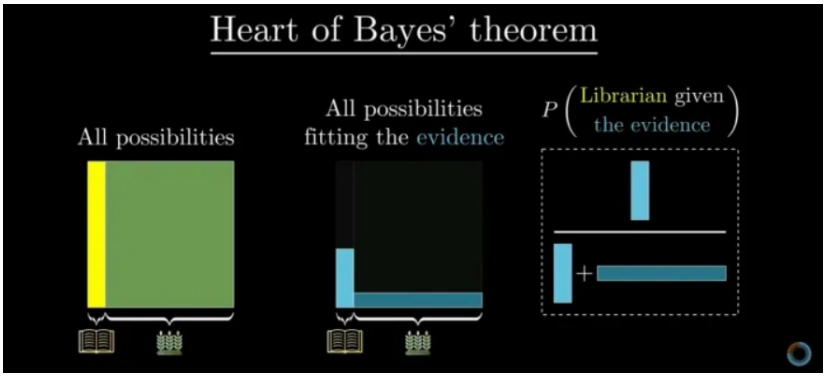

Figura B.1: Representación extraída de [11].

Si la hubieran tenido en cuenta, una forma de dar una estimación sobre la probabilidad de que Jaime fuese bibliotecario podría empezar por tomar una muestra aleatoria de granjeros y bibliotecarios, y hacer una estimación de cuántos de ellos podrían cuadrar con la descripción de Jaime.

Luego, la probabilidad de Jaime siendo bibliotecario se correspondería con el cociente entre el número de bibliotecarios con la personalidad de Jaime y el número de personas total con la personalidad de Jaime.

De manera que, en este ejemplo, aunque pienses que un bibliotecario es cinco veces más propenso que un granjero a tener esta personalidad, al final, el hecho de que hay muchos más granjeros que bibliotecarios acaba pesando más.

La moraleja es que pensar sólo en términos de que la descripción de Jaime se ajusta más a la de un bibliotecario no es suficiente para hacer una buena estimación, hay que tener en cuenta el hecho de que hay una proporción de granjeros/bibliotecarios de base. En otras palabras, nuevas evidencias no deben determinar por completo las creencias en un vacío: sino que deben de actualizar las creencias previas. Esta es precisamente la idea clave detrás del Teorema de Bayes, que la formaliza como la ec. (B.1).

Teorema de Bayes 1 *Sea* $\{A_1, A_2, ..., A_i, ..., A_n\}$ *un conjunto de sucesos mutuamente excluyentes y exhaustivos tales que la probabilidad de cada uno de ellos es distinta de cero* ($P(A_i) \neq 0$ *para* $i = 1, 2, ..., n$. *Si* B *es un suceso del cuál se conocen las probabilidades condicionales* $P(B|A_i)$, *entonces la probabilidad* $P(A_i|B)$ *viene dada por la expresión:*

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)} \tag{B.1}$$

donde $P(A_i)$ *son las probabilidades a priori,* $P(B|A_i)$ *es la probabilidad de* B *dada la hipótesis* A_i, *y* $P(A_i|B)$ *son las probabilidades a posteriori.*

En el anterior ejemplo hay dos hipótesis A_i, que Jaime sea granjero y que Jaime sea bibliotecario. El suceso B es la nueva evidencia, es decir, la descripción de la personalidad de Jaime. Releyendo la ec. (B.1) con la jerga del ejemplo se tendría:

$$P \text{ de que dada su personalidad Jaime sea bibliotecario} =$$

$$\frac{P \text{ de que siendo bibliotecario tenga esa personalidad} * P \text{ de que sea bibliotecario}}{P \text{ de que tenga esa personalidad con indiferencia de su profesión}}$$

Por otro lado, una aproximación frecuentista del problema directamente rechazaría trabajar con creencias y se fijaría únicamente en la proporción de granjeros/bibliotecarios para proporcionar una estimación sobre cuál sería la profesión de Jaime.

B.2 Regresión lineal con interpretación Bayesiana

Como se ha menciona en *Regresión lineal con interpretación frecuentista*, en dicha interpretación los θ son valores desconocidos, pero no aleatorios, y la dificultad radica en seleccionar la estrategia adecuada (como la de máxima verosimilitud) para estimarlos de la forma más precisa posible. En esta nueva interpretación se van a considerar los parámetros θ como variables aleatorias de valor desconocido. En primer lugar, se les asignará una distribución a priori $p(\theta)$ que exprese nuestras “creencias previas"sobre

los parámetros. Dado un conjunto de entrenamiento $S = \left\{(x^{(i)}, y^{(i)})\right\}_{i=1}^{m}$, si se quiere hacer una predicción sobre un nuevo valor de x se puede calcular la distribución a posteriori de los paramétros con la ec. (B.1):

$$p(\theta|S) = \frac{p(S|\theta)p(\theta)}{p(S)}$$

$$= \frac{\left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \theta)\right) p(\theta)}{\int_\theta \left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \theta)p(\theta)\right) d\theta}$$

Nótese que ahora escribimos $p(y^{(i)}|x^{(i)}, \theta)$, puesto que en este caso θ sí que es una variable aleatoria.

Reflexionemos un momento sobre esta interpretación. En el ejemplo de Jaime y su personalidad teníamos una evidencia previa, la proporción de granjeros y bibliotecarios, y la actualizábamos con la proporción de que en cada una de estas profesiones hubiese gente con la personalidad de Jaime. Intentábamos calcular cuál era la probabilidad de que Jaime tuviera una profesión u otra dada su personalidad. En este caso, intentamos calcular la probabilidad de los distintos valores de θ dado el hecho de que su función correspondiente ajusta a un conjunto de entrenamiento S. En esta analogía, los valores de θ serían las distintas profesiones, y el conjunto de entrenamiento S sería la personalidad de Jaime. Puesto que ahora tenemos que θ toma valores en un continuo (en contraposición a las profesiones, en cuyo caso solo considerábamos dos), ahora en vez de la probabilidad calculamos la densidad de probabilidad. Nuestra evidencia previa es la distribución de probabilidad que asignamos a los valores de los parámetros con indiferencia del conjunto de entrenamiento $p(\theta)$, y la actualizamos con cómo cada función asociada a cada uno de esos parámetros ajusta a los valores de S. Esto último es precisamente lo que calculábamos en la interpretación frecuentista: la verosimilitud.

Si en el ejemplo de Jaime decíamos que la moraleja era que, aunque su personalidad fuese mucho más común para un bibliotecario que para un granjero, al final acababa pesando más el hecho de que había muchos más granjeros que bibliotecarios, aquí diremos que, aunque un determinado valor de θ tenga una alta probabilidad de conseguir un mejor ajuste de los datos, si ese valor no es muy probable de entrada (antes de conocer el conjunto de entrenamiento), al final puede acabar pesando más esta baja probabilidad previa haciendo que elijamos otro valor como el más probable. Como se verá en la sección sobre *overfitting*, es precisamente esto en lo que nos apoyamos para que los modelos generalicen mejor.

En un acercamiento Bayesiano purista, cuando nos encontramos con un nuevo dato x y queremos hacer una predicción sobre él, calcularíamos la distribución a posteriori sobre la etiqueta y del dato utilizando la distribución a posteriori de los parámetros:

$$p(y|x, S) = \int_\theta p(y|x, \theta)p(\theta|S)\ d\theta$$

Así, por ejemplo, si se quiere calcular el valor esperado de y dado x se tiene que:

$$E[y|x, S] = \int_y y\ p(y|x, S)\ dy.$$

En la práctica, resulta que integrar sobre los θ, que suelen ser altamente dimensionales, no es en absoluto sencillo, así que se suele recurrir a un acercamiento más parecido al MLE que se utilizaba en la interpretación frecuentista.

Recordemos que en la sección frecuentista utilizamos la estrategia estadística de calcular el estimador de máxima verosimilitud (MLE); ahora vamos a hacer algo parecido pero maximizando la probabilidad a posteriori (MAP *Maximum a posteriori probability estimate*). Este acercamiento permite ignorar la

probabilidad $p(S)$, que como hemos dicho, en general es difícil de calcular debido a la dimensionalidad de los θ.

$$\begin{aligned}\hat{\theta}_{MAP} &= \arg\max_{\theta} p(\theta|S) \\ &= \arg\max_{\theta} \frac{p(S|\theta)p(\theta)}{p(S)} \\ &= \arg\max_{\theta} p(S|\theta)p(\theta).\end{aligned}$$

Y como hicimos en el caso de MLE, podemos maximizar el logaritmo por simplicidad:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} [\log p(S|\theta) + \log p(\theta)] \tag{B.2}$$

Nótese que el término del prior $P(\theta)$ no aparecía en el caso frecuentista. Realmente, los partidarios de esta interpretación critican la elección de un prior que no dependa de los datos, pero hay múltiples ocasiones en las que puede ser útil. Por ejemplo, cuando no se tienen muchos datos pero se conoce lo suficiente un problema como para tener una idea razonable sobre cómo ajustarlos. Incluso en casos con más datos se pueden elegir priors débiles, de manera que sea la verosimilitud $P(y|\theta)$ quien domine la ecuación para cambiar ligeramente el resultado de realizar MLE. De hecho, en cuanto a la verosimilitud se refiere, se hacen las mismas suposiciones que en el caso frecuentista y $p(y|\theta)$ acaba siendo la de la ec. (2.6).

En el caso de una regresión lineal, hay dos tipos de distribuciones probabilísticas a priori que se suelen considerar para asignar a los parámetros: la Normal y la Laplaciana.

Priors con Distribución Normal

Consideramos una distribución normal $N(0, \tau^2)$ de media cero y varianza τ^2 y consideramos $\theta \sim N(0, \tau^2 I)$, donde I es la matriz identidad con las dimensiones de θ. Haciendo las mismas consideraciones que se hicieron con el MLE, se puede asumir que en el caso de la regresión lineal la verosimilitud toma la forma de la ec. (2.6). Sustituyendo en la ec. (B.2):

$$\hat{\theta}_{MAP} = \arg\max_{\theta} [logP(y|\theta) + logP(\theta)]$$

$$= \arg\max_{\theta} \left[log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} \right\} + log \prod_{j=0}^{p} \frac{1}{\sqrt{2\pi}\tau} \exp\left\{ -\frac{\theta_j^2}{2\tau^2} \right\} \right]$$

$$= \arg\max_{\theta} \left[-\sum_{i=1}^{n} \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} - \sum_{j=0}^{p} \frac{\theta_j^2}{2\tau^2} \right]$$

$$= \arg\min_{\theta} \frac{1}{2\sigma^2} \left[\sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 + \frac{\sigma^2}{\tau^2} \sum_{j=0}^{p} \theta_j^2 \right]$$

$$= \arg\min_{\theta} \left[\sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 + \lambda \sum_{j=0}^{p} \theta_j^2 \right]$$

donde $\lambda = \frac{\sigma^2}{\tau^2}$. Por tanto, el efecto del prior será mayor o menor en función de la varianza. Distribuciones con una varianza pequeña aumentan el peso del prior en la suma (asignan probabilidades más desiguales a los distintos valores de θ, por tanto tiene sentido que tengan un mayor efecto sobre el cálculo de la probabilidad a posteriori). Por su parte, varianzas grandes disminuyen el prior (distribuyen la probabilidad más equitativamente), haciendo que la verosimilitud sea el término que domine la suma.

Priors con Distribución Laplaciana. Se utiliza una función de Laplace:

$$Laplace(\mu, b) = \frac{1}{2b} \exp\left\{-\frac{|x-\mu|}{b}\right\}$$

Nuevamente eligiendo una media de cero:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} \left[logP(y|\theta) + logP(\theta)\right]$$

$$= \arg\max_{\theta} \left[log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right\} + log \prod_{j=0}^{p} \frac{1}{\sqrt{2\pi}\tau} \exp\left\{-\frac{|\theta_j|}{b}\right\}\right]$$

$$= \arg\max_{\theta} \left[-\sum_{i=1}^{n} \frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2} - \sum_{j=0}^{p} \frac{|\theta_j|}{b}\right]$$

$$= \arg\min_{\theta} \frac{1}{2\sigma^2} \left[\sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 + \frac{2\sigma^2}{b} \sum_{j=0}^{p} |\theta_j|\right]$$

$$= \arg\min_{\theta} \left[\sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 + \lambda \sum_{j=0}^{p} |\theta_j|\right]$$

En este caso la Laplaciana concentra las probabilidades alrededor de su media, lo cual se traduce en un prior bastante fuerte que tienen un efecto *sparse* sobre los parámetros.

Apéndice C

Modelos secuenciales clásicos en NLP

C.1 Modelos secuenciales

Aunque son útiles, las técnicas de búsqueda de información tradicionales, como las basadas en *tf-idf*, tienen una lacra fundamental: carecen de cualquier tipo de sentido semántico. En el Capítulo 4 se ve un caso práctico donde se realiza una clasificación de productos de supermercado sobre una base de datos de etiquetas. En este caso, el problema se hace evidente con productos del tipo "Ambientador de vainilla", donde el algoritmo no es capaz de identificar que "vainilla" no es lo más relevante del producto a clasificar. En este punto, una de las opciones es intentar un acercamiento sintáctico, por ejemplo, tratar al producto como si fuera un sintagma nominal donde "ambientador" es el núcleo y "vainilla" es un complemento del núcleo. De esta forma se consigue una jerarquización que puede ayudar a ponderar los resultados. Pero, además de ser algo difícil de implementar, ¿qué sucede cuando en vez de "ambientador de vainilla" se tiene "ambientador vainilla" (sería lo más común en un entorno de producción real) ? Sin la preposición, el análisis sintáctico es más complicado. Lo ideal sería que el clasificador entendiese que "ambientador" es una palabra que jerárquicamente esta por encima de vainilla por su sentido en el lenguaje.

Otro ejemplo típico de la importancia del sentido semántico lo encontramos en el problema del *análisis de sentimientos*. La aproximación clásica del problema de clasificar *tweets* en positivos o negativos era utilizar diccionarios de, valga la redundancia, palabras positivas y negativas. Se buscaban las palabras que aparecían en cada *tweet* de cada diccionario, se sumaban sus puntuaciones y se obtenía una etiqueta. Nuevamente, el problema venía con oraciones con un sentido semántico más complejo como las oraciones irónicas del estilo "Este buscador es recomendable para perder el tiempo". El analizador de sentimientos leía la palabra "recomendable" y asignaba una etiqueta positiva a la oración.

En el Capítulo 2 se ha introducido la arquitectura de red neuronal más básica: la densa. Este tipo de red neuronal busca correlaciones entre sus *inputs* utilizando la "fuerza bruta". No tiene en cuenta, por ejemplo, si en la naturaleza los datos se relacionan de forma secuencial. La mejor manera que he encontrado de entender intuitivamente la diferencia entre un aprendizaje denso y uno secuencial es el ejemplo del abecedario. Si se le pide al lector recitar el abecedario desde la A, estoy seguro de que lo hará sin ningún problema. Si se le pide recitarlo desde la R, quizás le tomé un segundo empezar a escuchar en su cabeza la letanía de letras $R,S,T,...$ Sin embargo, ¿qué sucede si se le pide recitarlo al revés desde la Z? ¿o nombrar la letra en la décima posición? Por lo general, estas tareas son más complicadas porque el abecedario es una de esas cosas que aprendemos utilizando la técnica mental de la secuencialidad: no lo aprendemos de forma densa sino que cada letra es un interruptor que activa el recuerdo de la siguiente. La arquitectura computacional que refleja este tipo de técnica memorística es la de *red neuronal recurrente*.

Redes neuronales recurrentes (RNN) en NLP: análisis de sentimientos

Las redes neuronales recurrentes entrenadas para lenguaje natural todavía entran dentro de los modelos del lenguaje que se entrenaban para tareas especificas del procesamiento de lenguaje natural de forma supervisada. Por tanto, imaginarse un problema particular como el análisis de sentimientos facilita su explicación.

Nótese que en este ejemplo tenemos un *output* para toda la oración. Hay otros casos, por ejemplo el *etiquetado de entidades por palabra*, donde el *output* es distinto para cada palabra. En función de las dimensiones del *input* y del *output* tenemos los distintos tipos de redes neuronales de la Figura C.1.

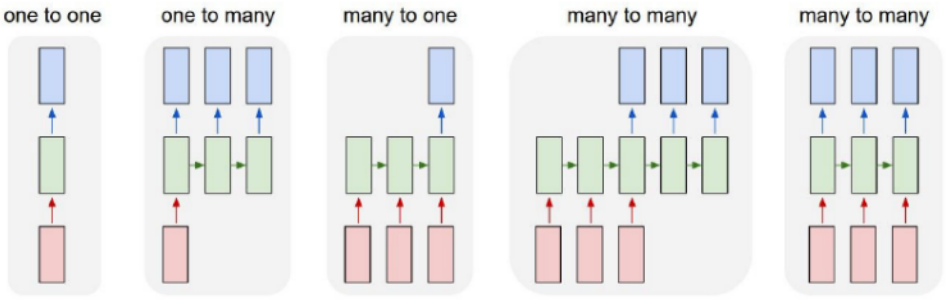

Figura C.1: Tipos de red neuronal recurrente (cs230 Lecture Notes Matt Deike)

En el caso del ejemplo se tiene una situación de *many to one*, así que la arquitectura de la red seguirá el esquema de la Figura C.2.

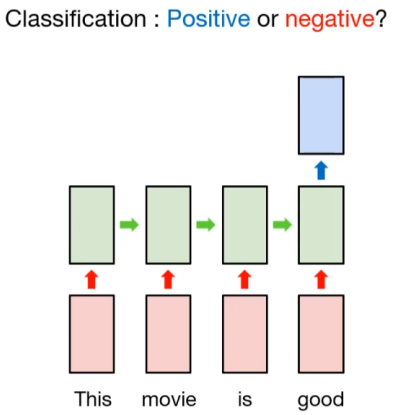

Figura C.2: Many-to-one RNN para análisis de sentimientos

Un matiz importante es darse cuenta de que el modelo de la red neuronal recurrente es un modelo que avanza en pasos temporales t, por tanto es un modelo direccional. Las palabras anteriores son las que influyen sobre las posteriores (en cualquier sentido que se elija). Esto es un problema con las palabras polisémicas, por ejemplo:

"Me senté en un banco antes de ir."

"Me senté antes de ir al banco."

Para solventar esto, más adelante se hablará de las redes recurrentes bidireccionales, por el momento se va a seguir explicando la unidireccional.

La idea de recurrencia en una red neuronal consiste en utilizar la información de pasos temporales anteriores. Esto se consigue calculando la activación de las neuronas de capas posteriores utilizando la activación de la capa anterior.

Formalicemos los conceptos mencionados. Consideramos como secuencia de entrada una frase x como *input* y un *output* y (en el caso de análisis de sentimientos una variable que toma valores en un espacio discreto de dimensión el número de sentimientos con los cuales etiquetar la frase. Por ejemplo, en la Figura C.2 se tendría un espacio de posibilidades binario: positivo o negativo. Supongamos que se *tokeniza* por palabras cara oración. Cada *token* del *input* se introduce a la red neuronal en un paso temporal t, y se denota por x^t. La longitud de la secuencia de entrada se denota por T_x y la longitud de la secuencia de salida por T_y (en este caso $T_y = 1$). Con $x^{(i)t}$ denotamos al *t-ésimo token* (en este caso la *t-ésima* palabra) del *i-ésimo input* y $T_x^{(i)}$ a su longitud. En cada paso temporal el modelo está parametrizado por los pesos compartidos W_{ax} y W_{aa}. Sea $g(x)$ la función de activación y b_a el *bias* (*valor umbral)* asociado.

$$a^t = g(W_{aa}a^{t-1} + W_{ax}x^t + b_a) \tag{C.1}$$

Cada capa de la red neuronal menos la última computa la ec. (C.1) (se suele tomar $a^0 = 0$). Por su parte, la última computa el *output* final con la ec. (C.2).

$$y = g(W_{ya}a^{t_f} + b_y) \tag{C.2}$$

A las capas encargadas de computar la nueva activación se las denomina *capas recurrentes* y a las de computar el *output* capas de *feed forward*. Normalmente la función de activación es ReLU para las capas recurrentes y *softmax* o *sigmoidal* para la capa final del output.

Backpropagation en el tiempo

Se considera la función de pérdida en cada paso temporal t de *entropía-cruzada*, o pérdida logística:

$$\mathcal{L}(y) = -y \log y - (1 - y) \log(1 - y) \tag{C.3}$$

Cuando se hace *backpropagation* en esta red se dice que se hace en el tiempo porque retrocede en la serie temporal. El proceso en este momento es el mismo que en cualquier algoritmo de aprendizaje automático: minimizar la función de pérdida y actualizar los pesos.

Desvanecimiento de gradientes con RNNs

Como ya se ha mencionado, las redes neuronales recurrentes para la modelización del lenguaje presentan dos problemas, principalmente: la dificultad en la paralelización y su corta memoria, esto es, su dificultad para contextualizar palabras muy separadas en la secuencia. Se debe a un problema conocido como el *desvanecimiento del gradiente*, habitual en aprendizaje profundo cuando se superponen muchas capas en la red neuronal. En el caso de las redes neuronales recurrentes aplicadas al lenguaje no es raro encontrarse con arquitecturas que superponen miles de pasos temporales t, con lo que se presenta este problema habitualmente.

***Long Short Term Memory* LSTM y GRUs**

Son los ejemplos de red recurrentes más famosos. Intentan solventar los problemas de las redes recurrentes con un mecanismo conocido como *gates*, pero, a pesar de conseguir mejorarlos, no llegan a resolverlos por completo.