TRABAJO DE FIN DE MÁSTER

Comparación de técnicas de optimización de modelos aplicadas al análisis de sentimientos.

Máster en Analítica del Negocio y Big Data
Universidad de Alcalá

Curso Académico 2024/2025

Alumno: Martín Iglesias Nalerio
Tutor: Antonio García Cabot
Cotutor: Pablo Trull Báguena

Contenido

1. Introducción y objetivos.

Hoy en día se generan ingentes cantidades de contenidos de múltiples tipos en Internet. Desde todo tipo de material audiovisual como canciones, películas o series, documentales, pasando por archivos de todo tipo como ficheros CSV, de texto plano, todos los archivos relacionados con la suite de Microsoft Office, hasta todos los ficheros fuente y binarios que hacen que los programas y aplicaciones informáticas que utilizamos diariamente funcionen.

La forma escrita es un método de comunicación y de expresión de, por ejemplo, hechos, ideas, pensamientos u opiniones, muy utilizado en Internet. A través de este método podemos expresar prácticamente todo tipo de sentimientos, exponer la realidad o nuestra percepción y opinión de la misma, entre otros muchos aspectos. Estas opiniones que generamos por nuestra parte o que obtenemos de terceros son muy valiosas en el mundo en el que vivimos, ya que nos influencian en lo que decidimos y llevamos a cabo diariamente. Desde qué objeto o servicio adquirimos, pasando por qué lugares visitamos en alguna escapada vacacional, hasta qué opinamos de las opiniones de otras personas, por ejemplo, en una red social, las opiniones son una parte fundamental del panorama actual.

Estas opiniones son tan poderosas que pueden provocar grandes fluctuaciones en el valor de las acciones de una empresa. Rápidamente se nos puede venir el caso de una empresa productora de películas que genere una nueva producción y que los usuarios estén o no estén encantados de la misma. Si hay una gran cantidad de opiniones a favor o en contra de la película, esto puede ayudar o perjudicar gravemente a la productora a la hora de obtener fondos de inversión para poder generar más películas o no.

Una película puede ser vista por millones de personas varias veces, y de esos visionados cada persona puede generar muchas opiniones que, por cuestiones obvias, las empresas no pueden leer al completo. Sin embargo, obtener la opinión mayoritaria de todos esos comentarios puede ser de gran utilidad, y en los tiempos que corren tenemos herramientas para obtener y analizar estas opiniones publicadas en Internet.

La inteligencia artificial (IA) definitivamente ha llegado para quedarse, y ha revolucionado la actualidad con sus múltiples utilidades. Los modelos de aprendizaje automático o profundo pueden procesar y aprender patrones (a priori invisibles) de grandes cantidades de datos, ayudando a mejorar procesos de diferentes campos. Otras utilidades de la IA pueden ser la visión artificial, detección de fraude, procesamiento de todo tipo de archivos multimedia, entre otras.

Dentro de todas las aplicaciones de IA, una muy importante es el procesamiento y análisis de textos. Utilizando técnicas de procesamiento de lenguaje natural (NLP), las máquinas pueden entender e interpretar el lenguaje humano. Con esto se pueden crear diferentes aplicaciones, desde chatbots que asistan al usuario en atención al cliente, pasando traducción, corrección y resumen de textos, hasta el

tema inicial de este trabajo, que es el análisis de gran cantidad de opiniones y la obtención del sentimiento del usuario.

El análisis de sentimientos en la actualidad necesita datos etiquetados, el texto original del cual se desea extraer la opinión, y la etiqueta que queremos que el modelo prediga, puede ser tan simple como si el sentimiento es positivo o negativo, o más preciso como enfado, tristeza o humor, entre otros. Estos datos se deben preprocesar adecuadamente para que sean la entrada de una red neuronal que prediga el sentimiento. Después de esto se puede evaluar el modelo y sus predicciones contra la realidad para obtener la precisión.

Estos modelos de base pueden ser grandes, de primeras deben aprender todo el vocabulario antes de empezar a inferir el significado. Esto hace que los modelos de análisis de sentimientos puedan ser intensivos en cuanto a la utilización de los recursos de procesamiento y de memoria, lo que puede ser un gran inconveniente ya que el modelo puede ser no utilizable según el dispositivo. Es muy importante encontrar un balance entre el tamaño y complejidad del modelo, y la precisión que provee. No queremos modelos super precisos ya que pueden estar sobre entrenados y no generalizar bien o ser muy complejos, pero tampoco queremos modelos sin suficiente entrenamiento que sean demasiados simples y no nos den suficiente rendimiento en las predicciones. Para este tipo de cuestiones existen técnicas que optimizan estos modelos, y la comparación de estas técnicas son la temática de este trabajo.

Existen diferentes técnicas de optimización de modelos de análisis de sentimientos, podemos resaltar como ejemplos las técnicas de cuantización, podado, destilación de conocimiento o adaptación de bajo rango (LoRA). Cuando realizamos este tipo de optimizaciones, el modelo preentrenado pasa por un proceso de ajuste fino (fine tuning) en el que el modelo se adapta a una nueva tarea, normalmente reentrenándolo con menos datos y ajustando ciertos parámetros.

Si queremos proveer brevemente el funcionamiento de las técnicas mencionadas, LoRA utiliza propiedades de matrices y de algebra para que las actualizaciones en el reentrenamiento se hagan con matrices de bajo rango, consiguiendo así reducir la complejidad del modelo al disminuir el número de parámetros entrenables. La destilación de conocimiento se aprovecha de un modelo de gran tamaño y precisión que enseña a un modelo de menor tamaño a predecir cómo este. Por su parte, el podado y la cuantización se aprovechan del tamaño de la red neuronal para asumir que hay partes de la misma, como algunos pesos o activaciones, que a efectos prácticos no contribuyen al rendimiento del modelo.

En este proyecto se evaluarán algunas de estas técnicas de optimización bajo los mismos datos y el mismo modelo base, con el objetivo de obtener conclusiones sobre el rendimiento de las mismas.

2. Encuadre teórico.

En este apartado sentaremos las bases teóricas y conceptuales de las técnicas utilizadas, con el objetivo principal de saber cómo funcionan, qué se está haciendo y qué se quiere obtener. El proyecto se encuadra dentro del marco del procesamiento del Lenguaje Natural (NLP, Natural Language Processing), así que comenzaremos por exponer y documentar sobre esta área y del tipo de redes utilizadas, así como la preparación de los datos necesaria para que estas redes funcionen adecuadamente, y la construcción del modelo que posteriormente se utilizará como base para evaluar las diferentes técnicas de optimización.

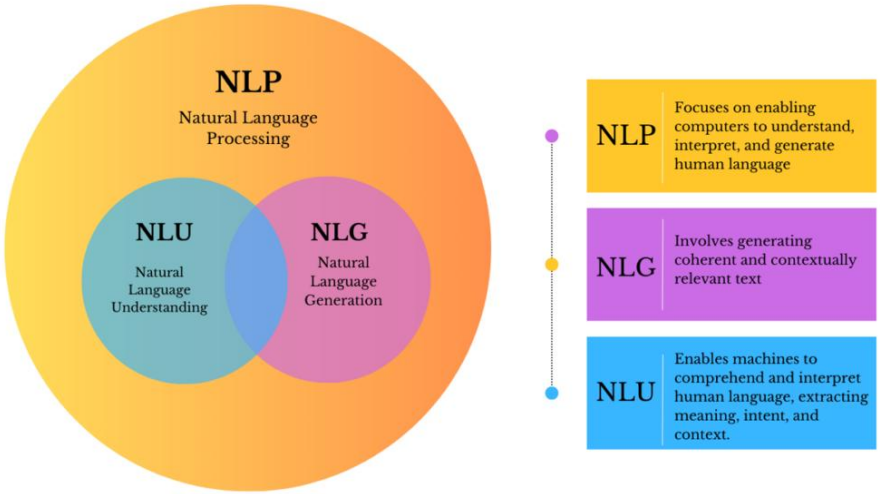
2.1 Procesamiento de Lenguaje Natural (NLP).

2.1.1 Definición.

El procesamiento del lenguaje natural (lenguaje natural hace referencia a la lengua humana) es un área dentro de la Inteligencia Artificial en el que se trata de que las máquinas puedan obtener como entrada de datos lo que se le dice a través de nuestro idioma, con el objetivo de comprenderlo y entenderlo para realizar acciones determinadas en base a ello. Este campo trata con datos en idioma humano, en diferentes formatos como por ejemplo escrito o hablado, los cuales son ricos en detalles debido al gran vocabulario que tenemos disponible, para crear aplicaciones y herramientas útiles de acuerdo a estos datos. Podemos poner como ejemplo cualquier asistente virtual de las grandes empresas tecnológicas, en donde verbalizamos una frase como “Oye Siri, enciende la luz.”, y el asistente toma eso como entrada, interpreta lo que queremos, y realiza una acción en base a ello, en este caso encender la iluminación.

NLP se puede subdividir en dos subpartes, el Entendimiento del Lenguaje Natural (NLU, Natural Language Understanding) y la Generación de Lenguaje Natural (NLG, Natural Language Generation). La primera se centra en dar significado a través de la semántica a la información que obtiene de los datos, apoyándose en la Inteligencia Artificial. Un ejemplo de esto serían el análisis de sentimientos, en donde obtenemos alguna forma de lenguaje humano y se distingue el sentimiento, emoción o la intención de estos datos. Por su parte, la segunda se centra en producir lenguaje natural a partir de datos estructurados. Uno de los ejemplos más clásicos de NLG podría ser los chatbots de atención al cliente en tiendas online.

Hoy en día las aplicaciones de NLP son amplias, y está presente en el día a día de cientos de millones de personas. Desde resumir, clasificar o traducir textos, hasta extraer información de los mismos y responder a preguntas, así como ya tener una conversación fluida, esta área juega un papel muy relevante en la sociedad, y aún queda trabajo por realizar para que estas herramientas funcionen adecuadamente en cualquier circunstancia, sin que devuelvan respuestas incoherentes o sesgadas.



La idea del NLP es que las máquinas entiendan el lenguaje humano y tomen acciones en base a lo que se les comunique. Sin embargo, hemos escuchado muchas veces que las máquinas hablan con ceros y unos, es decir, con lenguaje binario. Esto es completamente cierto y, por tanto, necesitamos alguna manera de traducir nuestro lenguaje humano a datos binarios que la máquina pueda entender. Esto se realiza a través de la tokenización, lo cual es trocear los datos en lengua humana en los denominados tokens, que serían un símil de las palabras en lenguaje humano, pero en formato vector.

2.1.2 Evolución temporal del NLP.

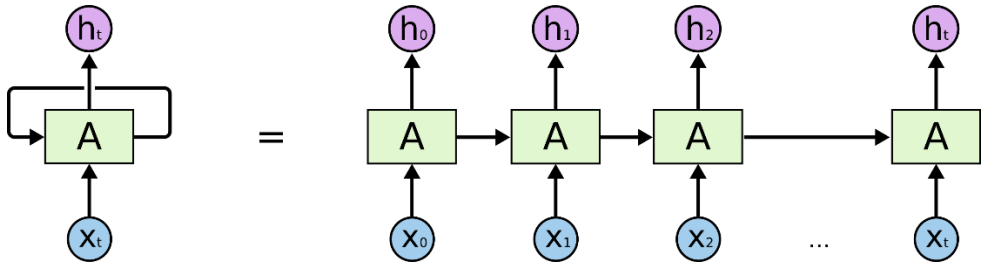
El NLP ha evolucionado a lo largo del tiempo, los primeros modelos se basaban en reglas condicionales sencillas antes de evolucionar a un NLP estadístico en donde se calculaba la probabilidad de la siguiente palabra o letra en base a la palabra anterior. Se utilizaban diccionarios para conocer si según la palabra encontrada el sentimiento era positivo o negativo, y se confiaba en reglas como que, por ejemplo, si se tiene la palabra Barack, muy probablemente la palabra que le siga sea Obama.

Con el impacto de la llegada de los modelos de aprendizaje automático y de aprendizaje profundo, esta área de NLP tuvo una gran mejora. Aparecieron distintos tipos de implementaciones utilizando redes neuronales y una mayor cantidad de datos, aunque estas maneras de realizar NLP no son aptas para todos los equipos. Dentro de este tipo de NLP con redes neuronales, algunos ejemplos destacan, tales como redes neuronales recurrentes (RNN), las redes LSTM (Long Short Term Memory), así como los Transformers y los Autoencoders.

El problema que tienen las redes neuronales tradicionales es que se entrenan desde cero, pero a la hora de predecir no tienen en cuenta lo que se ha dicho antes, ya que no tienen memoria, y esto es un inconveniente en el ámbito de análisis de sentimientos, ya que en un texto y de una oración a otra se puede cambiar el sentimiento y significado. Las redes neuronales recurrentes tratan de paliar este aspecto añadiendo cierta memoria al tener nodos que se conectan a sí mismos (son

nodos recurrentes) y que actúan como una memoria oculta que guardan ciertos elementos de la secuencia que han procesado previamente, haciendo que esa información persista en la red.

A continuación, se presenta una imagen de un nodo de una red neuronal recurrente:



Como vemos, el nodo admite una entrada X_t así como la memoria o estado oculto guardado hasta entonces h_t, y produce una salida h_t que se alimenta a otros nodos, es similar a si tuviésemos copias de los nodos que van calculando y actualizando su información en base a la entrada. La función de estas redes se expone a continuación, y se puede ver su recursión:

$$h_t = f(W_{hh} * h_{t-1} + W_{hx} * x_t + b_h)$$

En donde:

- h_t es la memoria o estado oculto actual.
- f() es una función matemática de activación no lineal, puede ser la tangente hiperbólica tanh.
- W son matrices de pesos en pasos anteriores y actuales. W_{hh} computa la capa oculta h_t dada la entrada oculta anterior h_{t-1}. W_{hx} introduce el vector de entrada X_t en el espacio del estado oculto.
- h_{t-1} es el estado oculto en el anterior paso.
- x_t es la entrada actual del paso.
- b_h es el sesgo actual del paso.

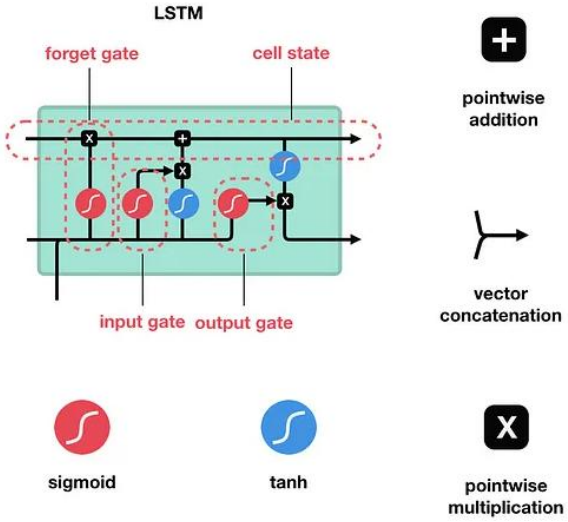
Estas redes recurrentes guardar cierta información previa, pero a veces no es suficiente para desempeñar la tarea que queremos, por ejemplo, si tiene una oración que indica que alguien habla inglés nativamente, no es capaz de deducir que esa persona nació en un país angloparlante. Esto es debido al problema que tienen este tipo de redes con los gradientes desvanecientes o explosivos.

Al entrenar una red neuronal se utiliza el método de retropropagación de errores para actualizar sus pesos y que la red vaya aprendiendo. En el caso de una red neuronal, este método se modifica ligeramente con redes recurrentes. El problema es que los gradientes se multiplican entre sí por cada recursión, y con mensajes de gran longitud puede suceder que exploten o se desvanezcan. Si estos gradientes tienen valores muy cercanos a cero el gradiente será cero y no podrá aprender de las últimas palabras ya que al retropropagar y llegar al principio se habrá “olvidado”

del final, el gradiente se habrá desvanecido. Si estos gradientes tienen valores muy grandes pueden producir unos pesos “explosivos” que no sean estables para entrenar el modelo.

Por estos problemas de las redes recurrentes simples es que surgieron las redes LSTM (Long Short Term Memory). Estas redes LSTM son un tipo de red neuronal recurrente que pueden aprenden de una secuencia más grande, debido al uso de un estado que va pasando por todos los nodos LSTM almacenando información, así como diferentes gates o compuertas que se utilizan para añadir o descartar información de este estado o memoria. Estos dos elementos hacen que el problema de los gradientes desvanecientes o explosivos que tienen las RNNs explicadas previamente se controlen y soluciona otros inconvenientes de las mismas, haciendo que las redes LSTM sean la base de este tipo de redes que deben aprender dependencias entre grandes cantidades de texto, y sean ampliamente utilizadas.

Las redes LSTM tienen diferentes variaciones (una de ellas son las redes GRU), las cuales tienen un rendimiento similar entre ellas. Sin embargo, la estructura de una LSTM simple tiene el siguiente aspecto:



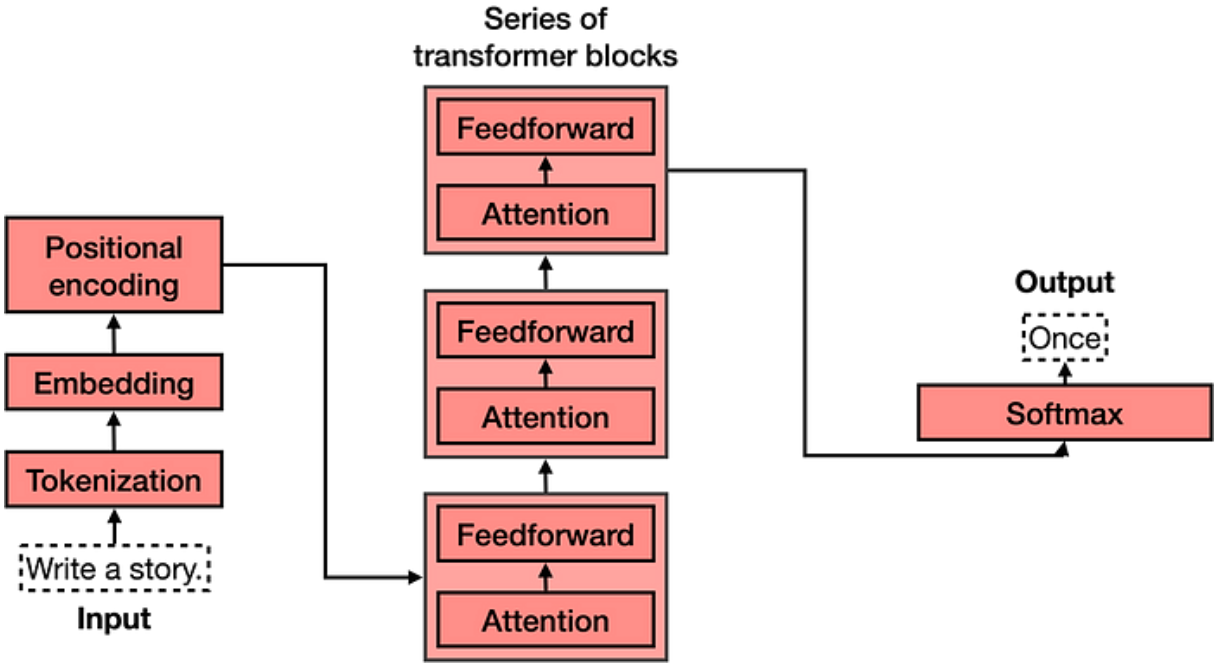
Cuando la información llega a la red LSTM, lo primero que hace es, a través de las compuertas de olvido o Forget Gates, y utilizando la función sigmoide, la cual devuelve valores entre cero y uno, se evalúa qué información se retiene (valor de sigmoide a uno, pasa adelante) y qué información se olvida (valor de sigmoide a cero, no pasa). Se utiliza esta función ya que los valores multiplicados por cero dan como resultado cero, ayudando a olvidar. En otras partes de la red se utiliza la función tangente hiperbólica (tanh), que tiene valores entre -1 y 1.

Después de esto, la red calcula la información que puede ser importante mantener para posteriormente actualizar la memoria, utilizando el estado anterior y la entrada actual. Estas dos fuentes de información se pasan por la función sigmoide, por una parte, para obtener la información candidata a actualizar, y por una función tangente hiperbólica para regular que los valores de la red estén normalizados. Luego de esto, estas dos salidas se combinan y con eso y lo que habíamos calculado de valores en la compuerta de olvido, tenemos el estado nuevo calculado.

Por último, en la compuerta de salida, se decide el estado de la memoria. Para ello, se combinan los datos del estado anterior y la entrada por una sigmoide, y los datos del estado nuevo calculado se pasan por la función hiperbólica, y estos dos resultados se multiplican para así decidir qué información tendrá la memoria. En definitiva, las compuertas de olvido sirven para descartar información no relevante, las compuertas de entrada deciden qué información puede ser relevante en el paso actual, y la compuerta de salida determina lo que se almacena en memoria y se pasa al paso siguiente.

El siguiente avance de este tipo de redes son los transformadores, y estos constituyen un gran hito ya que son utilizados en todo tipo de generadores multimedia, ya sea generar texto (como ChatGPT), generador de imágenes (como Dall-E 3) o de audio o video. Como curiosidad, las siglas GPT significan Transformador Generativo Preentrenado.

La estructura de un transformador se puede entender a través del siguiente gráfico:



Los transformadores son modelos de aprendizaje automático que utilizan el concepto de atención propia (self-attention) introducido en 2017 en un escrito científico de Google. En estos modelos, se tiene una entrada de texto y la primera parte es tokenizarlo (normalmente esto es dividir el texto en palabras), para luego convertir cada token en un vector numérico que representa su significado (llamado embedding), de tal manera que palabras con significado similar generan vectores similares. Luego de esto, estos vectores pasan repetidamente por una capa de self-attention en primer lugar, en donde los vectores comparten información entre sí para saber qué palabras son importantes en cuanto a significado, así como también para actualizar significados según el contexto (esto ayuda a diferenciar palabras homónimas, por ejemplo) y actualizan el significado; y una red neuronal en segundo lugar donde nos quedamos con patrones del texto y actualizamos los vectores según

el resultado de entrenamiento. Con esto, obtenemos en el último vector los valores que han ido condensando todo el significado del texto anterior. Sobre este vector final se realiza una operación softmax teniendo en cuenta todos los tokens (todo el vocabulario disponible) para obtener la probabilidad de la siguiente palabra del texto. Con esto se toma la palabra con mayor probabilidad (o a veces la segunda o tercera para crear variedad de respuestas) y se genera la siguiente palabra del texto. Este proceso se realiza varias veces hasta generar la respuesta completa deseada.

Con estos avances en los transformadores llegan los LLMs, o modelos extensos de lenguaje. Estos LLMs usan los transformadores previamente explicados para realizar su primera fase de preentrenamiento, en donde a los transformadores se les provee de ingentes cantidades de información para que aprenda a predecir la siguiente palabra. Para hacerse una idea, si un humano no parase de leer en todo el día, necesitaría miles de millones de años para leer toda la información provista a este modelo. Una vez el modelo ha sido preentrenado, pasa por una fase de ajuste fino en el que se le enseña a hacer lo que queramos que haga, en el caso de ChatGPT sería responder a lo que se le diga. Por último, se aplican técnicas de aprendizaje reforzado con retroalimentación humana para mejorar y ajustar aún más el modelo, utilizando a personas para que descarten respuestas erróneas o de poca utilidad, y premiando las respuestas apropiadas, para que el modelo obtenga una mejor idea de lo que se espera de este.

2.2 Técnicas de optimización.

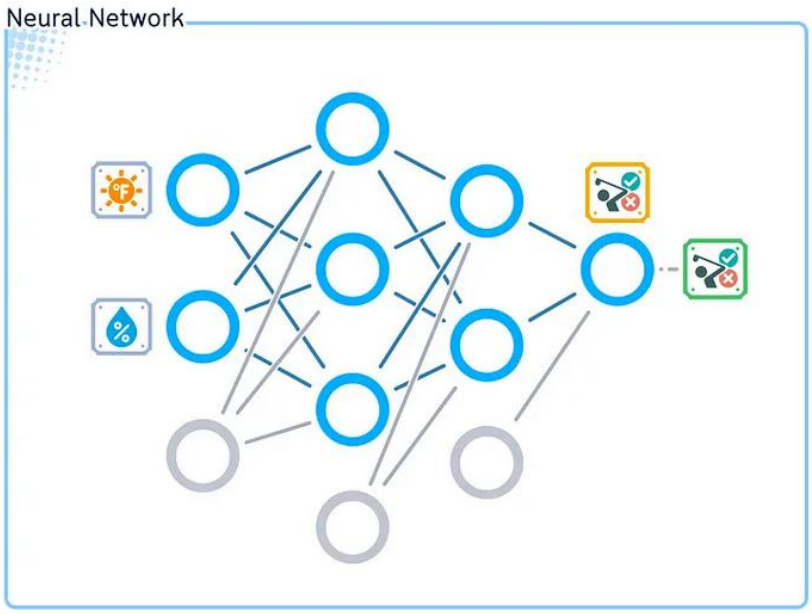
Con todo esto ya se ha ahondado en gran medida en el conocimiento necesario para este proyecto dentro del área de procesamiento de lenguaje natural. Los modelos de aprendizaje automático y profundo son de gran utilidad para resolver problemáticas de diferentes áreas, sin embargo, son computacionalmente intensivos y costosos y no cualquier dispositivo puede crear y utilizar estos modelos. De hecho, las grandes empresas tienen sus propios centros de datos y procesamiento, y firman contratos con suministradoras eléctricas y proveedoras de chips GPU (los cuales son de gran utilidad para realizar operaciones en paralelo, útiles por ejemplo en los transformadores que hemos visto) para abastecerse y ser capaces de proveer estos gigantescos modelos como servicio, a cambio de una cuota de suscripción. Para ello, y por otros motivos, existen técnicas de optimización con las que tratamos de hacer al modelo eficiente, es decir, que cumpla con los objetivos utilizando únicamente los recursos justos y necesarios.

Debido a que el campo de aplicación del aprendizaje automático es amplio, las técnicas de optimización están mayormente pensadas dependiendo el área o la aplicación de la red. En este caso el proyecto se centrará en las técnicas de optimización para modelos utilizados en análisis de sentimientos. De entre estos hay varios que podemos destacar, como la cuantización, el podado, la destilación por conocimiento o la adaptación de bajo rango (LoRA).

2.2.1 Cuantización.

Vamos a comenzar con la cuantización. La cuantización es una técnica de optimización de modelos de redes neuronales, en la que el objetivo es reducir el tamaño de la misma red tratando de perder rendimiento o precisión en ella. Obviamente, si a una red neuronal se le quita detalle perderá precisión, por tanto, la idea es encontrar un balance entre el tamaño y el rendimiento del modelo. La optimización no viene por la parte de que el modelo prediga mejor, si no por la parte de que conseguimos algo más pequeño que haga el mismo trabajo, lo que da lugar a que podamos crear modelos más grandes y precisos que luego se puedan cuantizar, reduciendo su tamaño y manteniendo la precisión.

En la cuantización se pueden reducir el tamaño de diferentes partes de una red neuronal. Como breve apunte, una red neuronal multicapa se puede imaginar con el siguiente aspecto:



Our network consists of 4 layers: 1 input layer (2 nodes), 2 hidden layers (3 nodes & 2 nodes), and 1 output layer (1 node). This creates a 2–3–2–1 architecture.

Y se compone de los siguientes elementos:

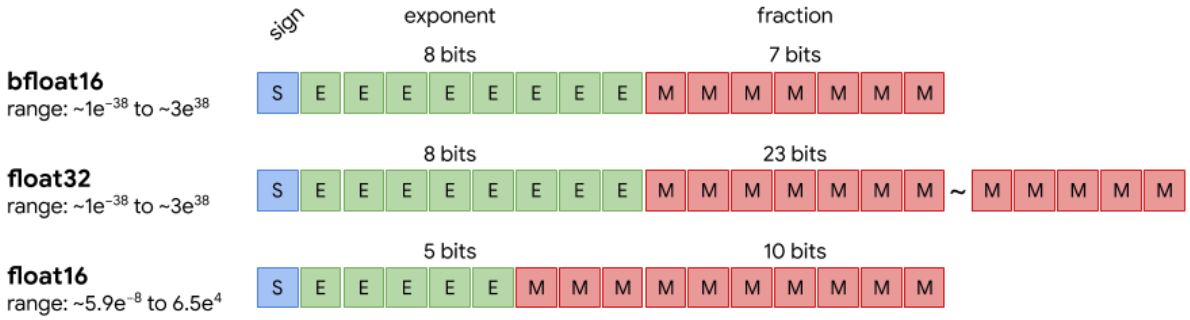
- Las neuronas o nodos son las unidades que componen la red, y que se agrupan en capas. La capa de entrada tiene tantos nodos como características tengamos, en la fotografía tiene dos entradas: la temperatura y el porcentaje de humedad.
- Las capas ocultas están entre la de entrada y salida, y sirven para detectar patrones complejos e ir progresando hacia el resultado.
- La capa de salida nos dará la respuesta que queremos, en este caso si podemos salir a jugar al golf dado los parámetros del clima.
- Los nodos se conectan entre sí con los pesos (W), que son números que indican que información es relevante. Son los parámetros que se ajustan durante el entrenamiento.

- El sesgo o bias es un valor adicional que permite que ajustar la red según nuestra conveniencia o necesidad.
- Las funciones de activación se aplican en cada neurona para permitir a la red aprender valores complejos. Toman los valores de los pesos y el sesgo y producen una salida.

Se pueden cuantizar los pesos y el sesgo, y las activaciones, entre otros aspectos menos comunes. Las redes neuronales utilizan una gran cantidad de vectores y matrices compuestos de valores numéricos, y la idea de reducir el tamaño de la red pasa por reducir el detalle o la precisión de los números utilizados. Esto hace que exista el llamado error de cuantización, ya que perdemos el detalle según los bits que utilicemos para los valores numéricos. El número pi se representa como 3 en INT8, como 3.141 en FLOAT16, y como 3.1415927 en FLOAT32, y a más detalle más bits necesarios para almacenar esa información y, por tanto, mayor tamaño del modelo.

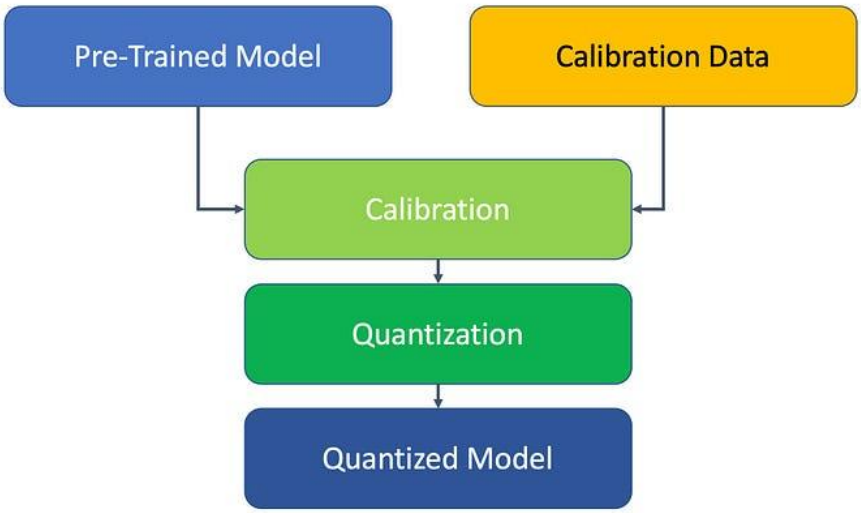
Para entender las diferencias entre las codificaciones, en primer lugar, INT8 es la representación más básica y comprende valores entre -128 y 127, utilizando 1 bit para el signo y 7 para el valor. La idea para obtener un valor en INT8 es eliminar la parte decimal si la tiene, convertir su valor positivo a binario, y en caso de ser negativo se invierten los valores de los bits y se le suma uno.

Las codificaciones de coma flotante tienen más cantidad de bits para almacenar decimales, y funcionan con tres componentes: el signo, el exponente y la precisión o mantisa. El signo siempre es 1 bit ya que o es positivo o es negativo. En el caso de FLOAT32 tenemos 8 bits para el exponente y 23 bits para la precisión, para la parte de FLOAT16 se tiene 6 bits para el exponente y 10 bits para la precisión. La gran mayoría de redes neuronales utilizan valores representados en FLOAT32.



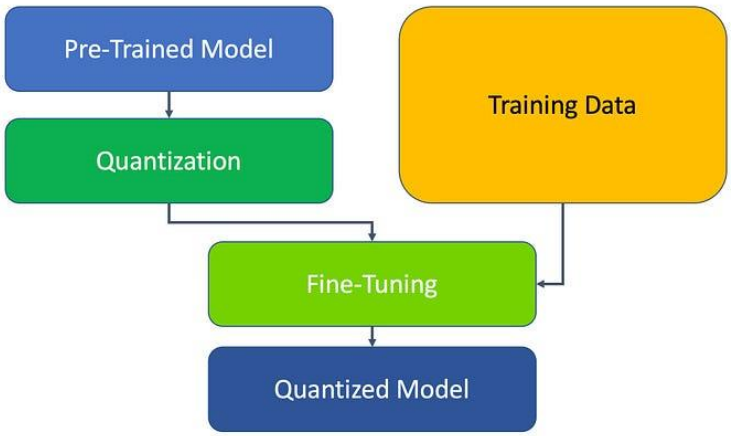
Hay diferentes formas de clasificar los tipos de cuantización dependiendo de qué se mire. Si se mira por el momento en el cual se aplica la cuantización, existe la cuantización después del entrenamiento (PTQ, Post Training Quantization) y la cuantización consciente del entrenamiento (QAT, Quantization Aware Training). En PTQ lo primero que se hace es entrenar el modelo con toda la precisión, en este caso utilizando FLOAT32, y después del entrenamiento se aplica la cuantización ya sea a pesos o a pesos y activaciones. Cuando se habla de aplicación de cuantización sobre pesos, se incluye pesos y el sesgo. Esta cuantización es rápida

de aplicar en comparación con su contrapartida, sin embargo, se puede perder más precisión.



Dentro de Post Training Quantization se pueden aplicar diferentes algoritmos: Cuantización de Rango Dinámico, Cuantización de Pesos o Cuantización por cada canal. En el primer algoritmo se cuantizan pesos y activaciones, pero únicamente para guardado en memoria, es decir, algunas partes como las activaciones siguen siendo representadas en FLOAT32. La cuantización se realiza observando el mayor y menor valor de los datos y teniendo en cuenta a qué tipo de dato se cuantiza. Esto reduce el tamaño del modelo, la inferencia es rápida, no requiere reentrenamiento, pero a cambio se puede perder precisión. En el segundo algoritmo únicamente se cuantizan los pesos, las activaciones permanecen representadas en FLOAT32, esto puede hacer que la precisión baje ya que los pesos cuantizados no funcionen bien con las activaciones no cuantizadas. Por último, el tercer algoritmo está más centrado en redes neuronales convolucionales (CNN), ya que estas tienen varios canales (suelen utilizarse para procesar imágenes, las cuales tienen tres canales, los canales de los colores RGB).

La alternativa es la cuantización consciente del entrenamiento (QAT), en la que la cuantización se realiza durante el entrenamiento el entrenamiento de la red, lo que en comparación con PTQ hace que pueda perder mayor precisión o detalle.



En este tipo de cuantización tenemos varios algoritmos: QAT con cuantizaciones falsas, QAT con estimador directo o QAT híbrido. En el primer algoritmo lo que se hace es introducir capas falsas de cuantización en el modelo, son falsas ya que simulan un redondeo hacia el tipo de datos que se cuantiza. En el segundo algoritmo se insertan estas capas y la cuantización solo se tiene en cuenta cuando los datos fluyen desde la capa inicial hasta la final (forward pass), en la retropropagación (backward pass) se sigue operando con representaciones en FLOAT32, lo cual lo hace más sencillo y rápido, pero menos preciso que QAT con cuantizaciones falsas. La tercera opción es, como su nombre indica, un híbrido entre las dos primeras donde el algoritmo decide en qué momento aplicar uno de los dos algoritmos.

Si miramos a qué es lo que se cuantiza, tenemos la Cuantización de Pesos y la Cuantización de Activaciones como las más comunes, y la Cuantización de Gradientes y Cuantización de Operadores como formas menos usuales. En la primera se cuantizan los pesos y los sesgos de las capas, y en la segunda se cuantizan las salidas intermedias entre capas, de hecho, estas dos se suelen combinar. La cuantización de gradientes cuantiza durante el entrenamiento estos elementos, y la cuantización de operadores hace referencia a utilizar operaciones más simples como sumas o multiplicaciones con datos con valores más restringidos o pequeños como INT8.

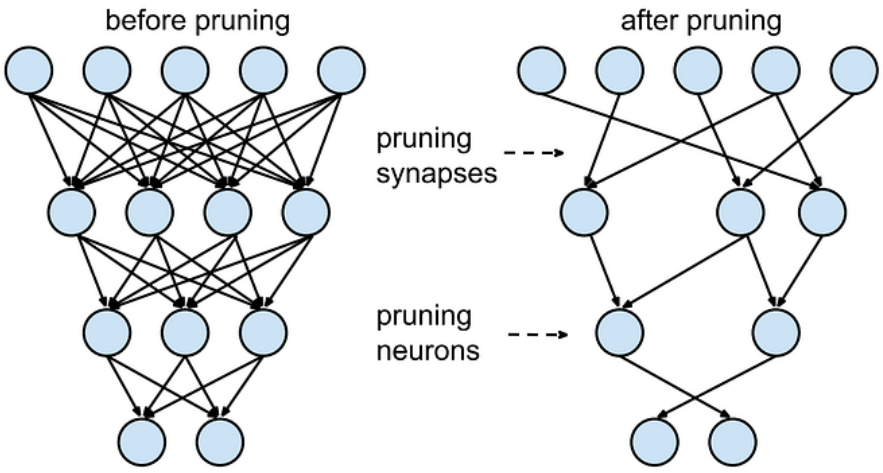
Si, por último, miramos a cómo se representan los valores cuantizados, tenemos la cuantización uniforme y la cuantización no uniforme. De la primera manera se toma el valor más grande y pequeño a cuantizar, y ese rango se divide en pasos iguales, los cuales hace que sea fácil de implementar. Sin embargo, de la otra manera podemos dividir ese rango de acuerdo a la distribución real de los datos, haciendo que sea algo más compleja pero más precisa.

Con todo esta esta información se pueden extraer ciertos aspectos. El primero de ellos es que la decisión del tipo de datos al que cuantizar es clave, no se puede reducir al máximo el tipo de datos ya que eso puede perjudicar más que ayudar. Si la tarea para la que se utilizará el modelo es precisa y sensible a cambios, necesitas detalle y por tanto FLOAT16 puede ser una buena opción. Si no es el caso, entonces se puede evaluar la opción de cuantizar a representaciones INT8. Dependiendo del tipo de cuantización, un proceso de ajuste fino con un breve reentreno puede ser muy importante. Por último, el benchmark o punto de referencia debe ser la precisión del modelo sin cuantizar frente al modelo cuantizado, teniendo en cuenta el tamaño en disco de los mismos.

2.2.2 Podado.

La segunda técnica de optimización que veremos en profundidad en este proyecto es la técnica de pruning o podado. La premisa de esta técnica es que hay algunas partes de una red neuronal que contribuyen entre nada y poco a la salida o predicción de la misma, ya que aportan un detalle muy sutil, y que si se eliminan estas partes se obtiene un modelo más eficiente y con una inferencia más rápida, sin

pérdida notable de rendimiento. Un ejemplo de cómo podría quedar una red neuronal habiéndole aplicado el podado es la siguiente:



Es importante no confundir la técnica de optimización del podado con la técnica de regularización de Dropout, ya que la primera sí afecta a la estructura del modelo, pero la segunda no. La regularización por Dropout se utiliza durante el entrenamiento para evitar el sobreentrenamiento y mejorar la generalización del modelo haciendo que cierto porcentaje de neuronas se desactiven temporalmente, la fase de inferencia no es afectada ya que ahí no aplica.

Hay diferentes formas de clasificar los diferentes tipos de podado. La primera de ellas es dependiendo de qué elementos se estén podando, y según esta clasificación podemos tener el Podado de Pesos y el Podado de Neuronas. Con el Podado de Pesos quitamos pesos o activaciones dentro de la red que se considere que no contribuyen al modelo. El hecho de determinar si un peso contribuye o no a la red se puede hacer a través del Podado basado en Magnitudes y el Podado basado en Gradientes, en el primer caso se observa si los pesos cuentan con magnitudes pequeñas (cercanas a cero) para eliminarlas, en el segundo se eliminan pesos para los cuales sus gradientes hayan llegado a un umbral (threshold) determinado en el que ya no vayan a variar mucho. También, dentro de esto, el Podado puede ser No Estructurado, eliminando conexiones entre neuronas de manera aleatoria y sin ningún tipo de patrón predefinido, o Estructurado, en el que se eliminan unidades enteras de la red de manera organizada para beneficiar las operaciones matriciales que se utilizan durante la etapa de inferencia. Esto significa que el Podado No Estructurado mantiene la estructura y forma de la red, ya que únicamente modifica pesos o activaciones, sin embargo, el Podado Estructurado sí que modifica la forma y estructura de la red.

Hay múltiples distintos métodos más para determinar si pesos o unidades de la red neuronal contribuyen con cierta relevancia al resultado. Por nombrar algunos, el Podado basado en el Porcentaje de Ceros elimina neuronas si con cierta consistencia devuelven valores similares a cero en el proceso de inferencia, es muy similar al Podado basado en Magnitudes; mientras, el Podado basado en Heurística

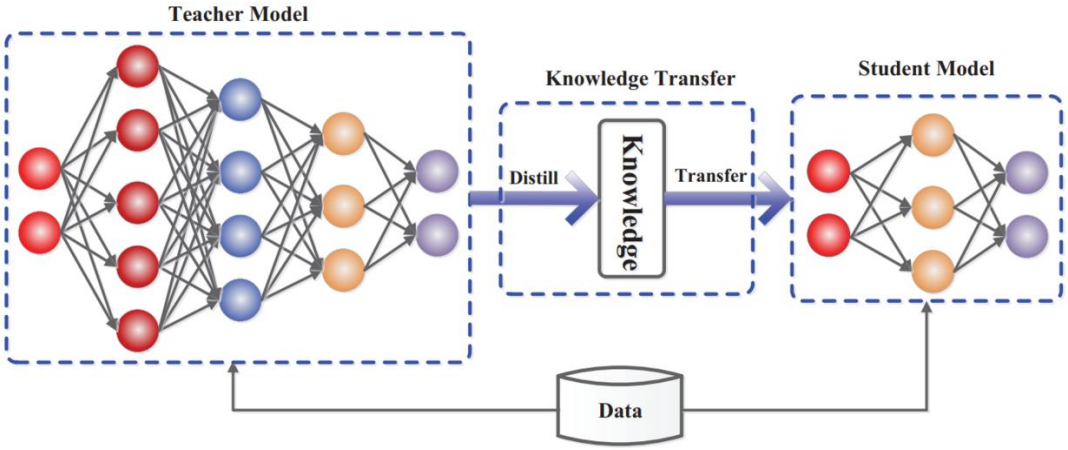
hace bastantes pruebas sobre cada capa de la red y con diferentes porcentajes de podado sobre la misma para ver cómo se deteriora el rendimiento, y tomar el porcentaje de podado medio que hace que el rendimiento no se degrade.

Por otra parte, se puede pensar que la técnica de podado siempre modifica la estructura de la red en la que se aplica, sin embargo, esto no siempre es así. Por ejemplo, en el Podado basado en Magnitudes a los pesos que tengan un valor cercano a cero se les pone una máscara con valor cero, de tal forma que no son tenidos en cuenta para la red, y a nivel estructural la red sigue viéndose y ocupando el mismo espacio en disco.

2.2.3 Destilación de Conocimiento.

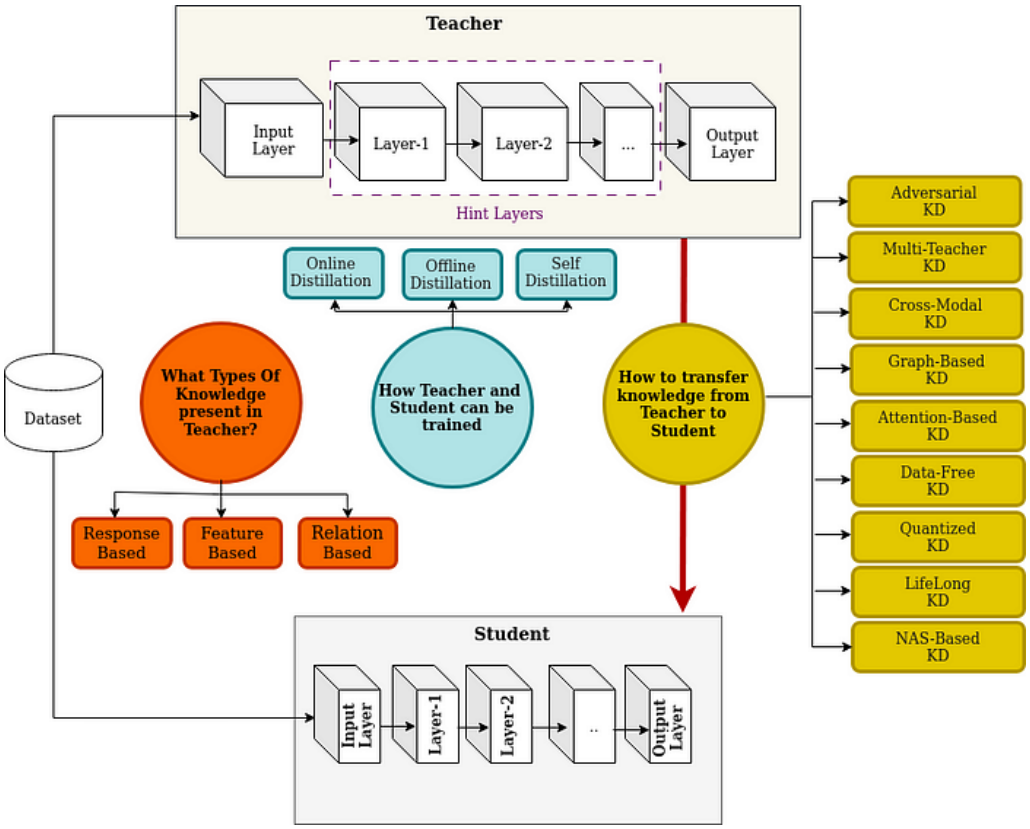
La última técnica en la que se entrará en detalle en este proyecto es la Destilación de Conocimiento (en inglés Knowledge Distillation). Se trata de una técnica dentro de la rama de aprendizaje automático en la que se tiene un modelo de gran tamaño y normalmente ya entrenado, y cuyo objetivo pasa porque ese modelo docente le enseñe o le traspase los conocimientos adquiridos a un modelo estudiante, así como "la forma de razonar", logrando de esta manera una reducción del tamaño del modelo sin alterar de sobremanera el rendimiento del mismo. Cabe resaltar que esta técnica ganó bastante popularidad después de que una empresa llamada Deepseek publicase de manera abierta (es decir, open-source) varios modelos extensos de lenguaje (LLM), entre ellos Deepseek R1, los cuales competían con los mejores modelos de lenguaje del momento en cuanto a rendimiento (siendo el más destacado el modelo de OpenAI) y, sin embargo, teniendo menor cantidad de parámetros y no necesitando chips de la mayor potencia posible para sus etapas de entrenamiento e inferencia.

Comúnmente la idea de la Destilación por Conocimiento es que modelos gigantes en cuanto a tamaño, con grandes capacidades y normalmente pertenecientes a grandes empresas se utilicen para transferir sus conocimientos en modelos más pequeños y accesibles y de código abierto, con el objetivo de hacer pruebas y mejorar las capacidades de este tipo de modelos. En la siguiente imagen se puede observar el aspecto estructural de la Destilación de Conocimiento:



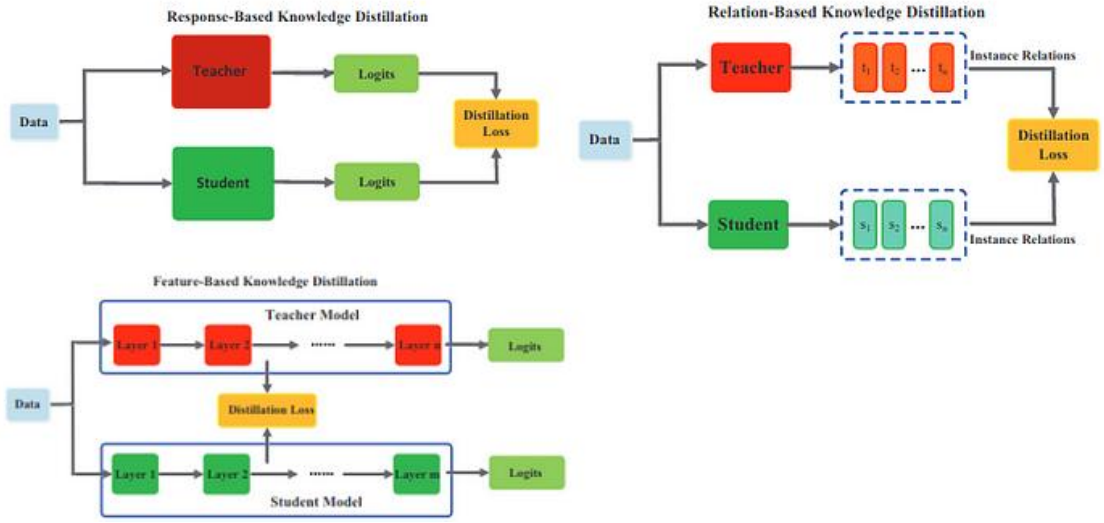
Al igual que con el resto de técnicas de optimización, la idea de ellas es que el rendimiento no es suficiente para elegir un modelo para la tarea que se quiera desarrollar, hay variables importantes como la cantidad de tiempo, recursos computacionales y dinero que hay que tener en cuenta. Los modelos con mayor precisión son muy grandes y consiguen ese rendimiento debido a esa combinación de tamaño y de cantidad y poder de procesamiento. Los modelos más modestos no son tan grandes, pero son más rápidos de entrenar, además de ser más explicables, a costa de perder precisión.

Hay diferentes maneras de clasificar los métodos de Destilación de Conocimiento. En la siguiente imagen se puede resumir las clasificaciones de este tipo de técnica de optimización:

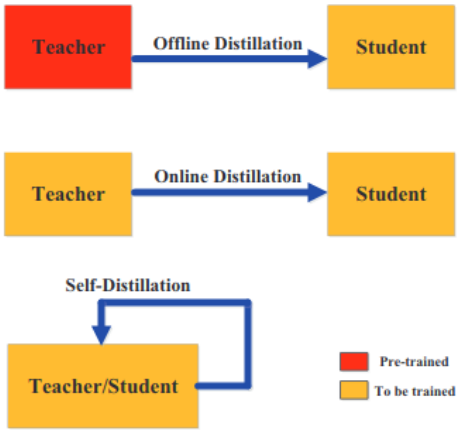


Se puede empezar por ver qué tipo de Conocimiento tiene el modelo docente y puede aprender el modelo estudiante. En este caso, del modelo docente se pueden estudiar las Respuestas, las Características o Features, y las Relaciones. La Destilación basada en Respuestas, que suele ser la más común debido a su buen funcionamiento en diversidad de tareas, se centra en entrenar al modelo estudiante con los datos de entrada, así como los resultados probabilísticos de la última capa del modelo docente, considerando las salidas como el conocimiento. La Destilación basada en Features o Características determina que el conocimiento no se encuentra únicamente en el resultado del modelo decente, sino que también se genera en las capas intermedias de la red, ya que en ellas el modelo docente detecta los patrones y el proceso real por el cual la red aprende a desempeñar su tarea, por lo tanto, el modelo estudiante no solo aprende el resultado que debe emular, sino que también aprende el proceso y los Features que el modelo docente

aprende y utiliza para llegar a dicho resultado. Por último, la Destilación basada en Relaciones trata de que el modelo docente aprenda y replique las conexiones entre las diferentes capas del modelo docente, siendo un método más complejo que los otros demás.



Los métodos de Destilación de Conocimiento pueden clasificarse según cómo se entrenan los modelos. En este caso tenemos la Destilación Desconectada, en la se puede tener y conservar un modelo docente previamente y, por tanto, se preentrena este modelo docente para, posteriormente, utilizarlo como base para entrenar el modelo estudiante. También está la Destilación Conectada en la que ambos modelos docente y estudiante se entrenan simultáneamente, ya que no es posible preentrenar el modelo docente con anterioridad por cualquier motivo, aunque esta técnica es más costosa computacionalmente. Por último, la Destilación Propia consiste en que tenemos únicamente un modelo que actúa tanto como docente como estudiante, se añaden ciertos elementos que hacen que la red aprenda de sus capas profundas, sin embargo, puede perder rendimiento con respecto a tener dos modelos docente y estudiante.



La idea para entender en qué consiste y cómo funciona la Destilación de Conocimiento hay que entender, en primer lugar, que no necesita que la red sea de ningún tipo, además de que el modelo docente y el modelo estudiante pueden ser

distintos tipos de modelos de aprendizaje; y, en segundo lugar, que las redes neuronales se consideran “aproximadores universales” ya que considerando suficiente cantidad de datos así como una estructura adecuada pueden aproximar su salida a la de cualquier función matemática con la precisión requerida.

Normalmente se suele utilizar Destilación de Conocimiento en tareas de clasificación, debido a que utiliza la función Softmax, la cual es muy útil en este tipo de tareas. La función Softmax es una función utilizada en inteligencia artificial para transformar los conocidos como logits, que son las puntuaciones que la red devuelve para cada posible resultado, en probabilidades. Esta función ayuda al modelo a decidir la clase pertenece un valor de entrada, asumiendo que la suma de probabilidades sume el valor de uno.

Después de esto, hay que tener en cuenta dos conceptos importantes. El primero de ellos es la Temperatura (T), la cual indica la confianza o certeza que se tiene en las predicciones realizadas. Este parámetro de Temperatura se tiene en cuenta en la función Softmax, siendo el valor $T < 1$ el que hace estar más confiados de la respuesta correcta (se le da una gran probabilidad, y al resto la probabilidad es muy cercana a cero, es poco flexible), $T = 1$ es el valor por defecto de la función Softmax, y $T > 1$ el que hace que la distribución de probabilidades se suavice, siendo el caso más común. Por hacer una comparación con una pregunta de un examen, una Temperatura más baja sería equivalente a un docente muy estricto en la corrección de esa respuesta, mientras que una Temperatura más alta sería similar a un docente más permisivo en dicha corrección. Con una Temperatura mayor se suavizan los resultados (Soft Targets), y estos se utilizan tanto en la destilación como en entrenamiento.

La idea es que el modelo docente genere esos logits que con Softmax y una alta Temperatura se convierten en los Soft Targets con los que el modelo estudiante entrena, además de con los valores verdaderos (Hard Targets), de tal manera que con los primeros aprende de los pasos que sigue el docente para hacer sus predicciones, y de los segundos aprende los resultados que debe dar. Con esto, se consigue que el modelo tenga una buena precisión, y al mismo tiempo generalice correctamente.

Las funciones de pérdida (Loss Function) en una red neuronal convencional hace referencia a un vector que representa la diferencia entre las salidas del modelo y las respuestas verdaderas dadas las entradas. La idea es que ajustando parámetros del modelo y utilizando algoritmos de optimización como el Descenso por Gradiente y la Retropropagación las salidas del modelo se acerquen lo máximo posible a las verdaderas respuestas. Sin embargo, pese a que esta función de pérdida es útil con los Hard Targets, no se utiliza para los Soft Targets, es decir, para evaluar qué tan bien el modelo estudiante aprende la forma de “razonar” del modelo docente. Para esta parte están las funciones de pérdida de destilación (Distillation Loss), con las que se evalúa esta este tipo de aprendizaje que debe hacer el modelo estudiante. Una función muy utilizada para esto es la función de Divergencia de Kullback-Leibler

(KL Divergence), por su buen funcionamiento en combinación con la función Softmax.

2.2.4 Tabla resumen.

Habiendo visto y desgranado en detalle algunas de las múltiples técnicas de optimización, se provee a continuación de una tabla resumen con una pequeña comparativa entre las técnicas descritas:

Técnica	Fundamental	Ventajas	Desventajas	Complejidad de la técnica	Tamaño del modelo	Velocidad de Inferencia
Cuantización	Reduciendo la precisión de los valores numéricos de la red neuronal se reduce el tamaño sin perder rendimiento.	Mantiene la complejidad y estructura del modelo reduciendo el tamaño entre 2 y 4 veces. Mejora la flexibilidad para implementar el modelo en hardware más limitado.	El resultado es posible que no funcione de manera óptima en el hardware (operaciones no soportadas), y no pueda ser acelerado para conseguir un rendimiento óptimo.	Baja	Mismo tamaño, pero menos detallado.	Rápida si se utiliza un tipo de dato soportado por el hardware donde se despliegue.
Podado	Eliminando partes de la red neuronal que no contribuyen de manera relevante al resultado, reducimos el tamaño sin perder precisión.	Reduce el tamaño del modelo manteniendo la estructura y la complejidad del modelo. Inferencia más rápida al no tener en cuenta partes del modelo.	La degradación del rendimiento puede hacer que sea necesario un proceso más largo de ajuste fino. Puede ser complejo de implementar de manera que se mantenga la eficiencia en el hardware.	Baja	Reducido.	Podado estructurado acelera el proceso, podado no estructurado no lo acelera.
Destilación de Conocimiento	Los grandes modelos son muy precisos, aunque costosos a nivel tiempo y dinero, por lo que se puede entrenar un modelo estudiante más pequeño que emule al modelo docente y conseguir un accuracy aceptable.	Se obtiene gran parte de la precisión de un modelo de mucha calidad a través de un modelo más pequeño que aprende a generalizar. Permite flexibilidad de estructura del modelo estudiante.	Se necesita un modelo de gran tamaño y preentrenado y, a veces, no es posible obtenerlo. Entrenamiento y obtención de hiperparámetros adecuados puede ser un proceso lento.	Media	Modelo estudiante reducido, modelo docente extenso.	Rápida para el modelo estudiante.

Con esto se han desglosado algunas técnicas de optimización, expuesto el detalle de su funcionamiento y comparado entre sí. A continuación, se expondrá el apartado práctico y el diseño metodológico del proyecto.

3. Diseño metodológico.

En este apartado se procederá a la implementación práctica de algunas de estas técnicas de optimización mencionadas. Para ello, se utilizará el mismo conjunto de datos (dataset) con el objetivo de entrenar una red neuronal base, a la que luego se le aplicarán estas técnicas de optimización para poder observar su rendimiento.

3.1 Entorno práctico.

En primer lugar, se ha decidido utilizar el lenguaje de programación Python, debido a que es un lenguaje de programación sencillo de aprender y de entender. Sin embargo, esta investigación práctica se puede llevar a cabo perfectamente en otros lenguajes relacionados, como pueden ser R o la plataforma KNIME. Para ser precisos, se utilizará Python 3.9 (no hay un motivo tras esta versión específica) y como entorno de desarrollo el editor de Microsoft llamado Visual Studio Code en su versión 1.100. Es importante que al instalar Python activemos la opción para añadirlo al PATH de Windows OS, que es el sistema operativo utilizado.

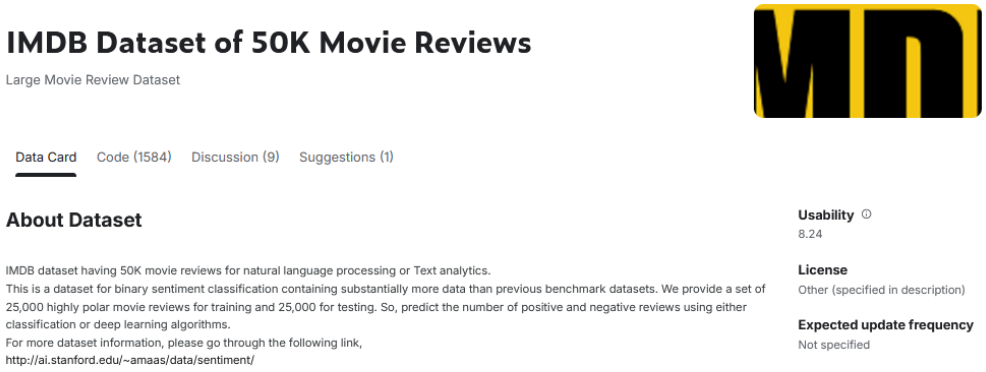
3.2 Obtención de datos del proyecto.

Lo primero y más importante a la hora de hacer cualquier trabajo con redes neuronales artificiales son tener unos datos adecuados. Para tener unos datos adecuados para una red neuronal, debemos realizar un preprocesamiento de los mismos que incluya una limpieza, normalización y adaptación de los mismos para el trabajo que se quiera realizar. Estos datos deben estar balanceados y ser una muestra representativa de todos los posibles valores, así como ser una buena cantidad para tener suficientes ejemplos con los que la red pueda aprender. Por poner un ejemplo, datos de cuatro mil millones de personas pueden parecer muchos datos, pero si esos datos son solo de mujeres pues no es representativo, ya que la sociedad humana no se compone únicamente de mujeres.

El conjunto de datos establecido para este trabajo es abiertamente conocido, y se trata de IMDB Reviews. Este conjunto de datos comprende, con mayor o menor detalle, diferentes críticas y opiniones a películas contenidas en la Base de Datos de Películas (Internet Movie DataBase, IMDB), así como el sentimiento que expresa dicha opinión en mayor o menor detalle, siendo la más básica si la opinión es

positiva o negativa. Otras versiones de este conjunto de datos incluyen más datos como el nombre de la película, el año de publicación, así como su duración o el presupuesto de la misma.

Este dataset se ha obtenido a través de la página web de Kaggle, donde se pueden obtener diferentes repositorios de datos:



3.3 Recolección y procesamiento de datos.

El código de Python para esta investigación se ha dividido en diferentes archivos separados, con el objetivo de dejar en claro cada parte del desarrollo. Se irá relatando los pasos realizados y se incluirán trozos de código en este documento que apoyen estos pasos. Los ficheros scripts de Python se adjuntarán también con este documento, de tal forma que puedan ser probados y ejecutados por el lector. Nos encontramos en la primera parte relacionada con la carga y preprocesamiento de los datos para su posterior explotación, y, por tanto, el primer script de Python que utilizaremos se llama “1-Data Collection and Preprocessing.py”, y contiene todo lo relacionado con este apartado del desarrollo.

En primer lugar, se necesitarán las siguientes librerías (instaladas previamente con el instalador de paquetes de Python llamado pip) para preprocesar los datos:

```
import pandas as pd
import re
import os
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from wordcloud import WordCloud
import matplotlib.pyplot as plt
```

Utilizando la librería de Pandas y su función head() podemos cargar el archivo con extensión CSV de IMDB Reviews y observar las filas de datos del principio del archivo que deseemos:

```
● CARGAR DATOS
                                                review sentiment
  0  One of the other reviewers has mentioned that ...  positive
  1  A wonderful little production. <br /><br />The...  positive
  2  I thought this was a wonderful way to spend ti...  positive
○ PS C:\Users\Martin>
```

Como podemos ver, ya en la segunda línea vemos Tags HTML que no aportan a los datos, e incluso pueden mermar el rendimiento. Para limpiar estos casos y, en general las críticas recibidas, se ha creado una función con la que normalizar los textos. El objetivo de la normalización es quitar signos de puntuación, espacios de más que pueda haber, y reducir la complejidad del texto de cara al modelo. La función es la siguiente:

```
# Normalizar texto.
def normalizar(texto):
    # Texto ya viene todo en minúsculas de la función de stopwords.
    # Quitar Enters.
    texto = re.sub('\n', '', texto)
    # Quitar tabulaciones.
    texto = re.sub('\t', '', texto)
    # Quitar HTML tags quitando todo lo que esté entre <>.
    texto = re.sub(r'<[^>]+>', '', texto)
    # Todo lo que no sea letras, números o apóstrofe lo ponemos a espa-
cios para luego reducirlos.
    texto = re.sub(r"[^\w\s']", ' ', texto)
    # Múltiples espacios de antes los pasamos a uno solo.
    texto = re.sub(' +', ' ', texto)
    return texto
```

Por otra parte, es importante en estos modelos de análisis de sentimientos eliminar las denominadas stopwords de los textos utilizados. Las stopwords son palabras dentro del texto que no aportan ningún tipo de información o no añaden valor. Por ejemplo, en el caso del texto “Un gato”, la palabra “Un” no añade gran valor al texto y se puede eliminar. Para ello, existen librerías en Python que descargan un listado de stopwords de un idioma, con ese listado limpiamos las críticas y opiniones de los datos de estas stopwords:

```
print('QUITAR STOPWORDS')
# Quitar stopwords: palabras que son muy comunes y que no añaden ningún va-
lor, como el 'Un' en 'Un móvil'.
nltk.download('stopwords')
#print(stopwords.words('english'))
```

```
ef quitar_stopwords(texto):
    # Obtener stopwords
    stop_words = stopwords.words('english')
    # Separar en palabras.
    palabras = texto.lower().split()
    # Aquí almacenaremos la review limpia.
    texto_limpio = ''
    # Por cada palabra de la frase, si es una stopword la quitamos, sino
la dejamos y mantenemos el espaciado.
    for palabra in palabras:
        if palabra not in stop_words:
            texto_limpio = texto_limpio + palabra + ' '
    return texto_limpio

# Al dataframe entero aplicarle la funcion, y ver cómo aplica en las tres
primeras líneas.
df['review'] = df['review'].map(quitar_stopwords)
print(df.head(3))
```

Aplicando estas funciones al conjunto de datos, y utilizando la función head() podemos ver los cambios:

```
QUITAR STOPWORDS
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Martin\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
                                              review sentiment
0  one reviewers mentioned watching 1 oz episode ...  positive
1  wonderful little production. <br /><br />the f...  positive
2  thought wonderful way spend time hot summer we...  positive
NORMALIZAR DATOS
                                              review sentiment
0  one reviewers mentioned watching 1 oz episode ...  positive
1  wonderful little production the filming techni...  positive
2  thought wonderful way spend time hot summer we...  positive
PS C:\Users\Martin>
```

A partir de esta limpieza podemos hacer un poco de Análisis Exploratorio de Datos (EDA) para observar la estructura de los mismos y determinar si debemos realizar alguna acción más. Se observará si hay datos nulos, duplicados, así como se obtendrán datos interesantes y descriptivos como la cantidad de palabras por crítica o la cantidad de caracteres únicos. El código utilizado para este apartado es el siguiente:

```
print('A BIT OF EDA')
# Un poco de EDA.
# Contar nulos en el dataframe.
print('CUANTOS NULOS')
print(df.isnull().sum())
# Datos genéricos del dataframe.
print('DESCRIPTIVO DATOS')
print(df.describe())
```

```python
# Palabras por review (nos vendrá bien para el diagrama Wordcloud)
print('CONTEO PALABRAS')
df['word_count'] = df['review'].apply(lambda x: len(x.split()))
print(df.head(3))
# Para cada sentimiento (positive, negative), sacar la media, mínimo y
máximo de palabras.
stats_Palabras = df.groupby('sentiment')['word_count'].agg(['mean',
'min', 'max'])
print(stats_Palabras)
# Añadir en una nueva columna la longitud en caracteres de la review.
print('LONGITUD CARACTERES REVIEW')
df['review_length'] = df['review'].apply(len)
print(df.head(3))
# Para cada sentimiento (positive, negative), sacar la media, mínimo y
máximo de caracteres.
stats_Longitud = df.groupby('sentiment')['review_length'].agg(['mean',
'min', 'max'])
print(stats_Longitud)
# Percentiles de 5 en 5, obtenemos las palabras y las contamos, y con
quantile sacamos los percentiles que queremos de ese grupo de documentos.
print("Percentiles del 1 al 100 en 5")
print(df["review"].str.split().str.len().quantile([i/100 for i in
range(0, 101, 5)]))
# Cantidad de caracteres únicos, juntamos todas las reviews en un texto
largo sin espacios, y lo transformamos en un conjunto (no admite dupps).
caracteres_unicos = set(''.join(df['review'].astype(str)))
print(f"Cantidad de caracteres únicos: {len(caracteres_unicos)}")
```

Al ejecutar las siguientes líneas de código obtenemos el siguiente resultado:

```
A BIT OF EDA
CUANTOS NULOS
review       0
sentiment    0
dtype: int64
DESCRIPTIVO DATOS
                                                   review sentiment
count                                               50000     50000
unique                                              49579         2
top     loved today's show variety solely cooking whic...  positive
freq                                                    5     25000
CONTEO PALABRAS
                mean  min   max
sentiment
negative   125.08472    3   875
positive   127.81848    6  1465
LONGITUD CARACTERES REVIEW
                mean  min   max
sentiment
negative   835.48688   18  5951
positive   869.16684   37  9394
Percentiles del 1 al 100 en 5
0.00       3.0
0.05      34.0
0.10      49.0
0.15      59.0
0.20      64.0
0.25      68.0
0.30      72.0
0.35      76.0
0.40      81.0
0.45      87.0
0.50      94.0
0.55     102.0
0.60     111.0
0.65     122.0
0.70     137.0
0.75     154.0
0.80     176.0
0.85     205.0
0.90     249.0
0.95     326.0
1.00    1465.0
Name: review, dtype: float64
Cantidad de caracteres únicos: 88
PS C:\Users\Martin>
```

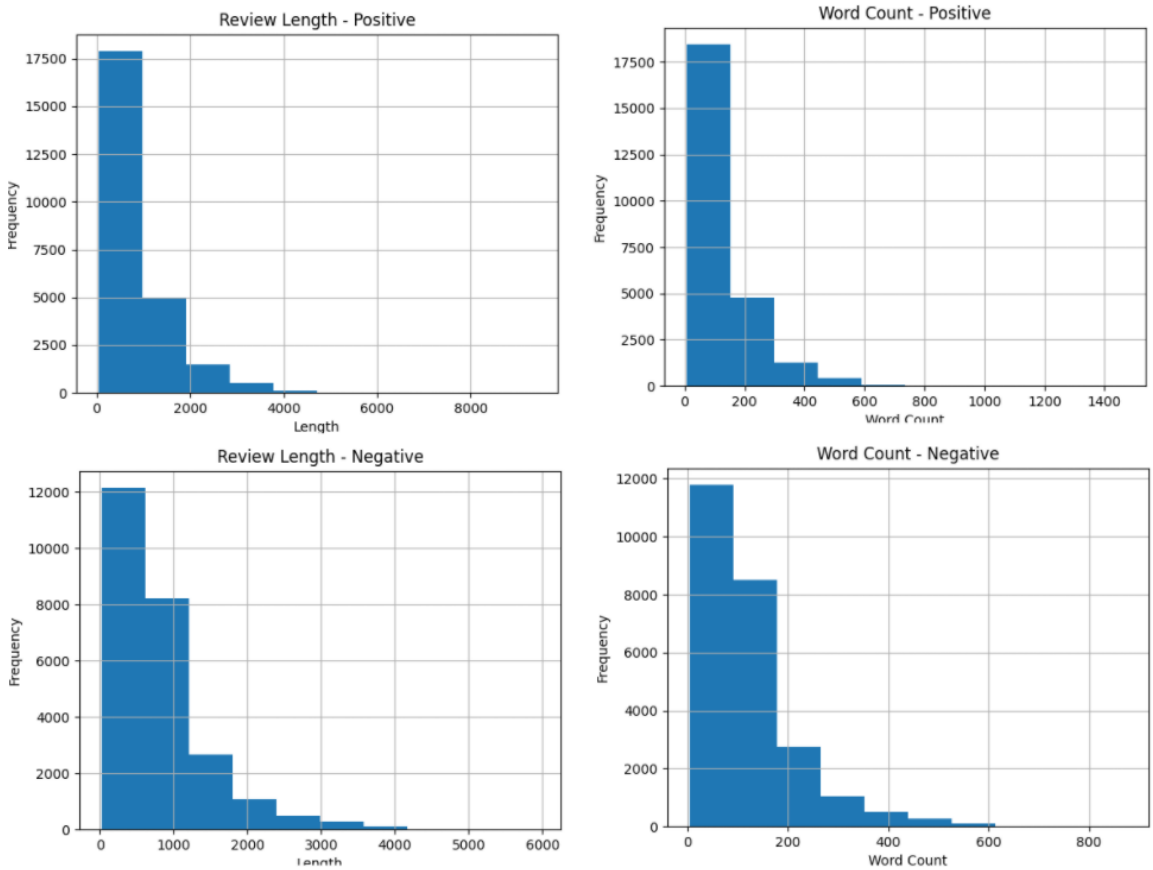
Como vemos en la imagen, podemos sacar las siguientes conclusiones sobre el conjunto de datos:

- No contiene ningún registro con valor nulo.
- Existen cincuenta mil filas para ambas columnas.
- La columna “sentiment” contiene dos únicos valores, “positive” o “negative”.
- Prácticamente la totalidad de las opiniones en la columna “review” son únicas.
- El conjunto de datos está balanceado, tenemos el mismo número de opiniones positivas y opiniones negativas.
- La media de palabras y caracteres por cada opinión es similar tanto para opiniones positivas como negativas. La opinión más extensa es positiva.
- La media de palabras por opinión está en alrededor de 125 palabras. Sin embargo, el valor mediano se sitúa en 94 palabras.
- El número de caracteres distintos encontrados son 88 caracteres.

Para observar esta información de una manera más cómoda, podemos generar diferentes gráficos que ayuden a interpretarlos de manera más rápida. A continuación, se ha utilizado el siguiente código el siguiente código para generar diferentes histogramas según el sentimiento de las opiniones:

```
# Gráficos para longitud de las reviews y cantidad de palabras
# Para cada sentimiento
for sentiment in ['positive', 'negative']:
    # Obtenemos los datos para ese sentimiento en un nuevo df.
    df_sent = df[df['sentiment'] == sentiment]

    print('HISTOGRAMA LONGITUD REVIEW: ' + sentiment)
    # Histograma para longitud de la review.
    df_sent['review_length'].hist()
    plt.title(f'Review Length - {sentiment.capitalize()}') # positive -->
Positive
    plt.xlabel('Length')
    plt.ylabel('Frequency')
    plt.show()

    print('HISTOGRAMA CANTIDAD DE PALABRAS REVIEW: ' + sentiment)
    # Histograma para cantidad de palabras de la review.
    df_sent['word_count'].hist()
    plt.title(f'Word Count - {sentiment.capitalize()}') # positive -->
Positive
    plt.xlabel('Word Count')
    plt.ylabel('Frequency')
    plt.show()
```

El resultado han sido los siguientes histogramas:



Por otra parte, podemos calcular el valor TFIDF de cada palabra y observar qué palabras tienen el valor más alto. El valor de TFIDF (Term Frequency – Inverse Document Frequency) mide qué tan importante es una palabra dentro de un documento (en este caso una opinión) y con respecto al resto de documentos evaluados. Es una técnica utilizada en NLP (Natural Language Processing), y con la que se mide cuántas veces aparece la palabra en el documento (Term Frequency) frente a la inversa de cuántas veces aparece en el resto de documentos (Inverse Document Frequency), es decir, si aparecen en todos los documentos IDF será bajo. Con esto conseguimos saber qué palabras son más relevantes sobre el conjunto de datos, aunque es importante que hayamos tratado las stopwords previamente.

Se ha utilizado el siguiente código para obtener las diez palabras con el TFIDF más alto:

```
# Obtener TFIDF (Term Frequency qué tan frecuente, IDF qué tan única es con
respecto a todos los documentos, más alto significa más importante o que
contiene/aporta mayor significado)
# Inicializamos el objeto que luego convertirá nuestros documentos en un
vector de números.
vectorizador = TfidfVectorizer(stop_words='english', max_features=1000)
```

```
# Matriz de filas los docs y columnas las palabras, valores son el valor
TFIDF.
matriz = vectorizador.fit_transform(df['review'])
# Para ponerle a las columnas el nombre de las palabras.
nombre_columnas_tfidf = vectorizador.get_feature_names_out()
# Pasar la matriz a un dataframe donde de columnas tenemos los features.
Pasamos las filas del doc a array ya aue la matriz es sparse (no guarda
ceros por eficiencia), así podemos convertir.
matrizDf = pd.DataFrame(matriz.toarray(), columns=nombre_columnas_tfidf)
# De cada feature sacamos el TFIDF medio, computandolo en base a su valor
en cada doc.
puntuaciones_medias = matrizDf.mean().sort_values(ascending=False)
# Obtenemos los 10 features con mayor TFIDF
print('FEATURES CON MAYOR SCORE DE TFIDF')
print(puntuaciones_medias.head(10))
```

El resultado de esta ejecución es el siguiente:

```
FEATURES CON MAYOR SCORE DE TFIDF
movie     0.084599
film      0.074423
like      0.041603
good      0.036216
time      0.030779
story     0.030717
really    0.030418
bad       0.028125
great     0.027587
people    0.025366
dtype: float64
PS C:\Users\Martin>
```

Vemos que destacan palabras habituales en críticas a películas como “película” o “historia”, y se hace evidente que el conjunto de datos está balanceado ya que palabras como “mala” o “buena” tienen un valor similar.

Por último, existen unos gráficos llamados Wordcloud, que consisten en una nube de palabras donde las más repetidas aparecen con mayor tamaño. Se ha utilizado el siguiente código para obtener estos gráficos:

```
# Wordcloud según sentiment.
# Obtener un string con todas las palabras de ese sentimiento separado por
espaacios.
palabras_negativo = ' '.join(df[df['sentiment'] == 'negative']['review'])
palabras_positivo = ' '.join(df[df['sentiment'] == 'positive']['review'])
```

```
# Generar gráficos Wordcloud
print('WORDCLOUD: NEGATIVE')
WordCloud(width=800, height=400, back-
ground_color='white').generate(palabras_negativo).to_image().show()
print('WORDCLOUD: POSITIVE')
WordCloud(width=800, height=400, back-
ground_color='white').generate(palabras_positivo).to_image().show()
```

Y los gráficos obtenidos son los siguientes, siendo el superior a la línea naranja para las opiniones negativas y el inferior para las opiniones positivas:

3.4 Creación del modelo básico.

Con esto pasamos al siguiente script en Python, llamado “2-LSTM Model.py”, en donde se construirá el modelo base que posteriormente se optimizará. Para este modelo las librerías importadas son las siguientes:

```
import pandas as pd
import re
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
from nltk.corpus import stopwords
from tensorflow.keras.preprocessing.sequence import pad_sequences
#from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
#from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

El modelo que se construirá es una red LSTM (Long Short Term Memory), por tanto, para ello, necesitamos adecuar los datos a la red y definir ciertos parámetros para construir la misma.

Utilizaremos la técnica de One-Hot Encoding para transformar los valores objetivo a una representación numérica, ya que las redes neuronales son modelos matemáticos y, por tanto, las cadenas de caracteres deben ser transformadas, además de evitar introducir al modelo relaciones innecesarias entre las palabras que puedan sesgarlo. En este caso, como solo tenemos dos clases, basta con asignar el valor 0 a negativo y 1 a positivo. Por otra parte, también las críticas se convertirán en vectores numéricos, pero para ello tendremos que traducir el vocabulario disponible a números. Por último, la división de los datos se ha establecido una división estándar de un 80% (40000 opiniones) para entrenamiento y 20% para datos de test.

Lo siguiente es definir ciertos parámetros y valores importantes para el modelo, esos valores son los siguientes:

- El tamaño del diccionario, es decir, cuántas palabras conocerá el modelo. En este caso tomaremos las 20000 más comunes en el idioma inglés.
- La longitud máxima de los documentos, en este caso se ha elegido 170 palabras según un proceso de pruebas realizado.

- El tamaño del batch, en este caso se ha escogido 64. El batch es la cantidad de documentos que se procesan a la vez antes de que el modelo actualice los valores de sus pesos.
- El número de épocas o epochs, en este caso se han escogido 12. Los epochs son cuántas pasadas hace sobre todo el conjunto de datos durante la etapa de entrenamiento.
- La posición de truncado y de “padeo” o relleno, si los documentos no llegan o sobrepasan las 170 palabras habrá que rellenar o truncar dicho documento, y no es lo mismo añadir o truncar desde el principio o desde el final del documento. En este caso se ha elegido hacerlo desde el final ya que parece la opción más intuitiva.
- El tamaño de las capas de la red, en este caso se han elegido 128 neuronas.

Se estableció un banco de pruebas con el objetivo de obtener la mejor configuración para el modelo en parámetros como el tamaño del batch, la longitud máxima de documentos, el tamaño de la red neuronal y el número de épocas necesario. Por otra parte, ya que la idea con esto es obtener un modelo al que luego le aplicaremos técnicas de optimización, ante dos configuraciones similares se escogerá la que requiera más recursos, con el objetivo de más adelante optimizarla. Las anotaciones realizadas sobre este banco de pruebas se pueden ver a continuación, comparando su rendimiento en el entorno de test y el máximo rendimiento conseguido en el entorno de entrenamiento:

- epochs10, dim64, maxLen70, batch32 --> Test accuracy 8020: 0.8263, 0.99
- epochs10, dim64, maxLen100, batch32 --> Test accuracy 8020: 0.8489, 0.99
- epochs10, dim64, maxLen130, batch32 --> Test accuracy 8020: 0.8650, 0.99
- epochs10, dim64, maxLen170, batch32 --> Test accuracy 8020: 0.8707, 0.99
- epochs10, dim64, maxLen230, batch32 --> Test accuracy 8020: 0.8715, 0.99
- epochs10, dim64, maxLen250, batch32 --> Test accuracy 8020: 0.8772, 0.98
- epochs10, dim64, maxLen310, batch32 --> Test accuracy 8020: 0.8816, 0.95

- epochs10, dim64, maxLen70, batch64 --> Test accuracy 8020: 0.8368, 0.99
- epochs10, dim64, maxLen100, batch64 --> Test accuracy 8020: 0.8531, 0.99
- epochs10, dim64, maxLen130, batch64 --> Test accuracy 8020: 0.8618, 0.99
- epochs10, dim64, maxLen170, batch64 --> Test accuracy 8020: 0.8647, 0.99
- epochs10, dim64, maxLen230, batch64 --> Test accuracy 8020: 0.8767, 0.99
- epochs10, dim64, maxLen250, batch64 --> Test accuracy 8020: 0.8627, 0.98
- epochs10, dim64, maxLen310, batch64 --> Test accuracy 8020: 0.8739, 0.97

- epochs10, dim128, maxLen70, batch32 --> Test accuracy 8020: 0.8338, 0.99
- epochs10, dim128, maxLen100, batch32 --> Test accuracy 8020: 0.8530, 0.99
- epochs10, dim128, maxLen130, batch32 --> Test accuracy 8020: 0.8641, 0.99
- epochs10, dim128, maxLen170, batch32 --> Test accuracy 8020: 0.8693, 0.99
- epochs10, dim128, maxLen230, batch32 --> Test accuracy 8020: 0.8770, 0.99
- epochs10, dim128, maxLen250, batch32 --> Test accuracy 8020: 0.8681, 0.99
- epochs10, dim128, maxLen310, batch32 --> Test accuracy 8020: 0.8728, 0.99

- epochs10, dim128, maxLen70, batch64 --> Test accuracy 8020: 0.8297, 0.99
- epochs10, dim128, maxLen100, batch64 --> Test accuracy 8020: 0.8541, 0.99
- epochs10, dim128, maxLen130, batch64 --> Test accuracy 8020: 0.8658, 0.99
- epochs10, dim128, maxLen170, batch64 --> Test accuracy 8020: 0.8723, 0.99
- epochs10, dim128, maxLen230, batch64 --> Test accuracy 8020: 0.8690, 0.99
- epochs10, dim128, maxLen250, batch64 --> Test accuracy 8020: 0.8699, 0.99
- epochs10, dim128, maxLen310, batch64 --> Test accuracy 8020: 0.8735, 0.98

```
# Creando modelo
# Las redes neuronales de Keras trabajan con números mejor que con textos.
Usaríamos un Onehot o Label Encoding para transformar a números.
print("ONE HOT ENCODING")
df['sentiment'] = df['sentiment'].map({'negative': 0, 'positive': 1})
 División de los datos 80/20 para la Red Neuronal LSTM, ponemos un random
state para replicar en el resto de scripts, y mezclamos para evitar algun
posible sesgo de datos (ejemplo: los 25k positivos y luego 25k negativos).
print("DIVISION DATOS")
x_train, x_test, y_train, y_test = train_test_split(df["review"],
df['sentiment'], test_size=0.2, random_state=1, shuffle=True)

# Parámetros para el modelo
print("DEFINICION DE PARÁMETROS PARA EL MODELO")
tamanio_diccionario = 20000
longitudMax = 170 # Longitud mediana de las reviews.
batch = 64
epochs = 12
posicion_truncado = 'post'
posicion_padeo = 'post'
palabra_oov = '404'
tamaño_capa = 128
```

Por otra parte, el código utilizado que define estos parámetros es el siguiente:

```
print("CONVERSION DE REVIEWS A LISTAS DE NÚMEROS APTAS PARA LA RED NEURO-
NAL")
# Creamos objeto que aprenderá el vocabulario.
tokenizador = Tokenizer(num_words = tamanio_diccionario, oov_token = pala-
bra_oov)
# Aprender el vocabulario.
```

Se ha construido una función en Python que devuelve un modelo LSTM. En primer lugar le indicamos que generaremos una red neuronal de varias capas secuenciales, siendo la primera una capa de Embedding que contiene el vocabulario que se utilizará transformado en vectores de números, seguida de una capa de células LSTMs y una capa de regularización por Dropout con la que cierto porcentaje de las neuronas se desactivan durante el proceso de entrenamiento con el objetivo de prevenir el sobreaprendizaje, y finalizando en una capa de una única neurona con activación sigmoide en la que se verá si el valor es una probabilidad más cercana a cero (opinión negativa) o más cercana a uno (opinión positiva). Por último, al compilar el modelo se utilizará el optimizador ADAM que controla la tasa de aprendizaje para cada parámetro del modelo (ya que si la ratio es muy alto perderemos detalle debido a problemas con los gradientes), y la función de pérdida de entropía cruzada binario, ideal para activaciones sigmoides ya que calcula la diferencia entre la probabilidad estimada (valor entre 0 y 1) y el valor real (0 o 1), y la red tratará de minimizar esa diferencia.

El código de la función de creación del modelo es el siguiente, y además podemos llamar a la función summary() que nos devuelve la estructura del modelo creado:

```
# Construir el modelo
def crear_modelo_LSTM():
    # El modelo será una sucesión de capas de neuronas.
    modelo = Sequential()

    # Primer set de capas.
    # La primera capa construye los embeddings, genera matriz que contiene
filas (vectores) para cada palabra.
    modelo.add(Embedding(tamanio_diccionario, tamaño_capa, input_length =
longitudMax))
    # La segunda capa tendrá las neuronas de memoria LSTM.
    modelo.add(LSTM(tamaño_capa))
    # La tercera capa es de Regularización por Dropout, el 20% de las neu-
ronas los ponemos a cero para evitar sobreaprendizaje en el entrenamiento.
    modelo.add(Dropout(0.2))

    # La última capa sigmoide nos dará probabilidad, si cerca de 1 será po-
sitivo, si cerca de 0 será negativo.
    modelo.add(Dense(1, activation = 'sigmoid'))
```

```
# Aglutinamos todas las capas y definimos cómo se evalúa el modelo.
    # Función de pérdida entropía cruzada binaria, funciona bien con cla-
sificaciones binarias frente a MSE.
    # Optimizador ADAM, robusto.
    # Queremos que cuando se entrene nos vaya indicando la preci-
sion/accuracy sobre train y sobre validación.
    modelo.compile(optimizer = 'adam', loss = 'binary_crossentropy', met-
rics=['accuracy'])

    return modelo

# Obtener nuestro modelo.
print("CONSTRUIR EL MODELO")
modelo = crear_modelo_LSTM()


# Resumen de la estructura de nuestro modelo, nos da información de capas
y parámetros.
print("RESUMEN ESTRUCTURA MODELO")
modelo.summary()
```

Si se ejecuta este código podemos obtener por pantalla el resumen mencionado del modelo:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 170, 128)          2560000

 lstm (LSTM)                 (None, 128)               131584

 dropout (Dropout)           (None, 128)               0

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
_________________________________________________________________
PS C:\Users\Martin>
```

La capa de Embedding tiene el tamaño del vocabulario multiplicado por el número de unidades de neuronas, en este caso 20000 * 128 = 2560000. Los parámetros de la capa LSTM se calculan con la fórmula 4 * (n * m + n^2 + n), en donde tanto n y m son las dimensiones de los vectores de entrada y salida de la capa, y que tiene que ver con las dimensiones de entrada a cada una de las compuertas que componen las células LSTM. En ambos casos son 128 unidades, así que la cuenta es 4 * (128 * 128 + 128^2 + 128) = 131584 parámetros. La capa de Dropout solo desactiva neuronas, entonces no es que tenga neuronas en la capa. La capa densa tiene 128

unidades de entrada y 1 salida, sumando el sesgo (mismo tamaño que salida) tendríamos tamaño_entrada * tamaño_salida + sesgo = 128 * 1 + 1 = 129 parámetros.

A partir de aquí ya podemos entrenar el modelo, evaluarlo con los datos de test, y sacar diferentes gráficos y estadísticas para observar su rendimiento fácilmente. Además de esto, guardaremos el modelo con sus pesos en un archivo para, posteriormente, poder cargarlo e ir aplicándole las técnicas de optimización. El código utilizado para todos estos pasos es el siguiente:

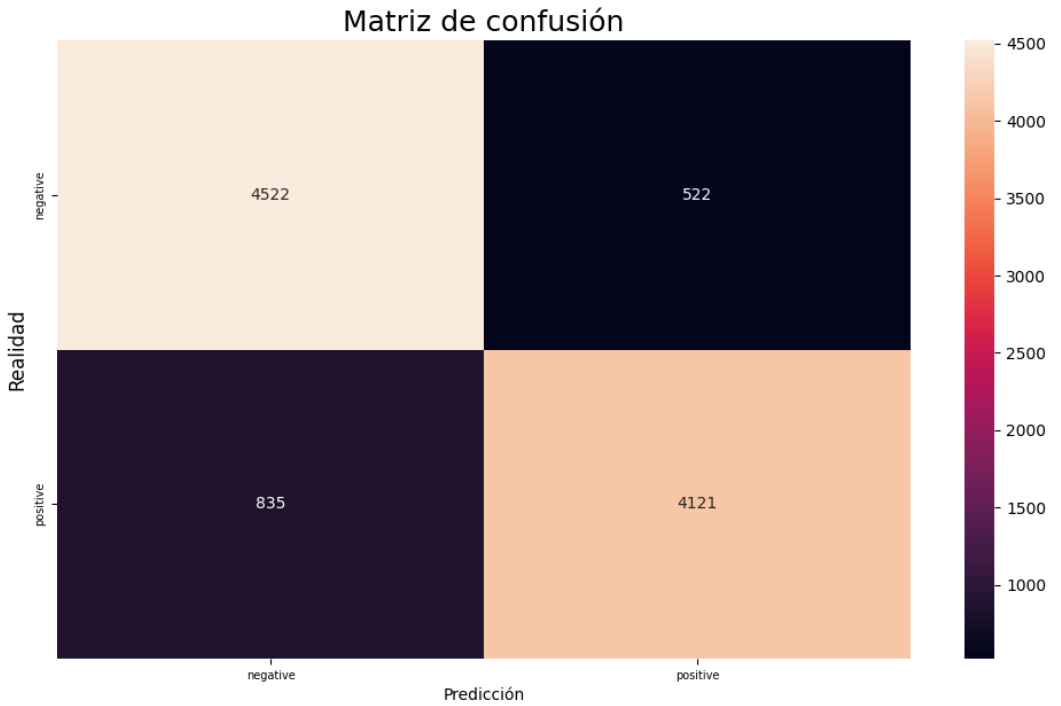
```
# Entrenar nuestro modelo compilado con datos de train, 10 vueltas a todos
los datos y procesa batch muestras a la vez mientras entrena.
# Muchos epochs pueden sobreentrenar, si más batch es más rapido pero nece-
sita memoria/procesamiento y si menos batch va más lento.
# Guardamos los resultados del entrenamiento en modelo_entrenado.
print("ENTRENAR MODELO")
modelo_entrenado = modelo.fit(X_train, y_train, batch_size = batch, epochs =
epochs, validation_split = 0.1)

# Gráfico de líneas con la evolución de la precision/accuracy.
print("GRÁFICO ACCURACY TRAIN VALIDATION")
plt.plot(modelo_entrenado.history['accuracy'])
plt.plot(modelo_entrenado.history['val_accuracy'])
plt.title('Precisión del modelo.')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

# Comprobar el modelo con los datos de Test, asegurando que sean vectores.
# Obtenemos en una lista la pérdida y la accuracy, como se especificó en
Compile().
print("COMPROBAR MODELO CON DATOS DE TEST")
evaluacion = modelo.evaluate(np.asarray(X_test), np.asarray(y_test))
print(f"Loss: {round(evaluacion[0], 4)}, accuracy:
{round(evaluacion[1],2)*100}%")

# La predicción nos dará valores entre 0 y 1 por la salida sigmoide, lo re-
dondeamos a 0 negativo o 1 positivo.
print("OBTENIENDO PREDICCIONES")
prediccion_probabilidades = modelo.predict(X_test)
# Del bool de si es > 0.5 nos dará true/false, que es 1/0 con el astype(), y
que con flatten convetimos la lista de listas a lista.
prediccion = (prediccion_probabilidades > 0.5).astype(int).flatten()
realidad = y_test
```

```
print("MATRIZ DE CONFUSIÓN")
matriz = confusion_matrix(realidad, prediccion, labels = [0, 1]) # Añadimos
labels para saberlas clases que se clasifican.
plt.figure(figsize = (12, 7))
graf = plt.subplot()
# Generamos la matriz con el valor en cada celda de la matriz (anot) en
formato entero (fmt), en el plot lienzo (ax) y no otro, y sustituir el 0, 1
con los labels.
sns.heatmap(matriz, annot = True, ax = graf, fmt = 'g', xtick-
labels=["Negative", "Positive"], yticklabels=["Negative", "Positive"])
# Establecer los labels para la información del gráfico, y guardar el grá-
fico en el workdir antes de mostrarlo.
graf.set_xlabel('Predicción')
graf.set_ylabel('Realidad', size = 12)
graf.set_title('Matriz de confusión', size = 18)
graf.xaxis.set_ticklabels(["negative","positive"], size = 7)
graf.yaxis.set_ticklabels(["negative","positive"], size = 7)
plt.savefig('matriz_confusion.png')
plt.show()

print("INFORME DE CLASIFICACIÓN FINAL")
informe = classification_report(realidad, prediccion)
print(informe)

print("GUARDANDO MODELO")
modelo.save('modelo_base.keras')
print("MODELO GUARDADO CORRECTAMENTE")
```

Este apartado ha sido lento a la hora de ejecutarse, el resto de códigos son prácticamente instantáneos, sin embargo, este código ha tardado 30 minutos en ejecutarse. Estos han sido los resultados que han aparecido por consola de comandos:

```
ENTRENAR MODELO
Epoch 1/12
563/563 [==============================] - 86s 150ms/step - loss: 0.6860 - accuracy: 0.5355 - val_loss: 0.6836 - val_accuracy: 0.5515
Epoch 2/12
563/563 [==============================] - 85s 151ms/step - loss: 0.6663 - accuracy: 0.5782 - val_loss: 0.6837 - val_accuracy: 0.5393
Epoch 3/12
563/563 [==============================] - 88s 156ms/step - loss: 0.6642 - accuracy: 0.5774 - val_loss: 0.7183 - val_accuracy: 0.6077
Epoch 4/12
563/563 [==============================] - 89s 159ms/step - loss: 0.5797 - accuracy: 0.6969 - val_loss: 0.6601 - val_accuracy: 0.6230
Epoch 5/12
563/563 [==============================] - 89s 158ms/step - loss: 0.5944 - accuracy: 0.6736 - val_loss: 0.5850 - val_accuracy: 0.6995
Epoch 6/12
563/563 [==============================] - 90s 160ms/step - loss: 0.5758 - accuracy: 0.6978 - val_loss: 0.5083 - val_accuracy: 0.7613
Epoch 7/12
563/563 [==============================] - 88s 157ms/step - loss: 0.4044 - accuracy: 0.8323 - val_loss: 0.3627 - val_accuracy: 0.8482
Epoch 8/12
563/563 [==============================] - 89s 157ms/step - loss: 0.2316 - accuracy: 0.9171 - val_loss: 0.3132 - val_accuracy: 0.8802
Epoch 9/12
563/563 [==============================] - 89s 158ms/step - loss: 0.1584 - accuracy: 0.9480 - val_loss: 0.3489 - val_accuracy: 0.8788
Epoch 10/12
563/563 [==============================] - 104s 184ms/step - loss: 0.1031 - accuracy: 0.9700 - val_loss: 0.3898 - val_accuracy: 0.8755
Epoch 11/12
563/563 [==============================] - 98s 174ms/step - loss: 0.0677 - accuracy: 0.9824 - val_loss: 0.4198 - val_accuracy: 0.8740
Epoch 12/12
563/563 [==============================] - 95s 168ms/step - loss: 0.0490 - accuracy: 0.9892 - val_loss: 0.4914 - val_accuracy: 0.8695
GRÁFICO ACCURACY TRAIN VALIDATION
COMPROBAR MODELO CON DATOS DE TEST
313/313 [==============================] - 9s 29ms/step - loss: 0.5069 - accuracy: 0.8643
Loss: 0.5069, accuracy: 86.0%
OBTENIENDO PREDICCIONES
MATRIZ DE CONFUSIÓN
INFORME DE CLASIFICACIÓN FINAL
              precision    recall  f1-score   support

           0       0.84      0.90      0.87      5044
           1       0.89      0.83      0.86      4956

    accuracy                           0.86     10000
   macro avg       0.87      0.86      0.86     10000
weighted avg       0.87      0.86      0.86     10000

GUARDANDO MODELO
MODELO GUARDADO CORRECTAMENTE
```

Y, entre otros gráficos, hemos obtenido la matriz de confusión, la cual luce de la siguiente manera:



Con esto hemos obtenido un modelo que arroja una accuracy del 86% con los datos de test, y con el resto de valores provistos en el informe final vemos que se trata de un modelo balanceado y adecuado para la tarea a realizar.

3.5 Cuantización.

Ahora, entramos en la parte de los scripts para aplicar técnicas de optimización sobre este modelo base. La primera técnica en la que nos centraremos es la técnica de cuantización, a través del script con nombre “3-Quantization.py”.

Lo principal en estos scripts de implementación de técnicas de optimización es importar el modelo entrenado guardado, de tal manera que no tengamos que volver a entrenarlo y pasar por ese costoso proceso. Para ello, el código utilizado es simple, y es el siguiente:

```
# Cargando modelo base.
print("CARGANDO MODELO")
modelo = keras.models.load_model('modelo_base.keras')
print("MODELO CARGADO CORRECTAMENTE")
```

Se van a realizar la implementación de dos formas de cuantizaciones, la cuantización de rango dinámico (Dynamic Range Quantization) y la cuantización al tipo de datos FLOAT16. En este caso, se definirán primero estos modelos, y posteriormente se evaluarán.

Para el caso de Dynamic Range Quantization, se utilizará la librería TFLite, la cual es una sublibrería de Tensorflow para dispositivos hardware más limitados, con el objetivo de convertir el modelo original importado en un modelo optimizado. El modelo original se asignará a un objeto conversor, en el cual se quitarán otros tipos de optimizaciones y se permitirá al conversor utilizar las operaciones tanto de TFLite como de Tensorflow, esto es así debido a las características que tiene el modelo original utilizado. Una vez se tiene definido esto, se puede realizar la ejecución de la conversión del modelo. El código que resume todo este desarrollo es el mostrado a continuación:

```
# <-------------------------------DYNAMIC RANGE QUANTIZATION DRQ-----------
-------------------->


# TFLite es Tensorflow para hardware más limitado, limita las funciones que
tenemos.
# Importamos el modelo base guardado.
conversor_DRQ = tf.lite.TFLiteConverter.from_keras_model(modelo)
# Dynamic Range Quantization (DRQ) no necesita un representative_dataset ya
que es Post Training Quantization, no Quantization Aware Training. Así que
no se lo pasamos.
# Aplicamos la optimización por defecto (la mejor que pueda) sin datos re-
presentativos. Se puede optimizar para mejor latencia/inferencia, tamaño...
conversor_DRQ.optimizations = [tf.lite.Optimize.DEFAULT]
```

```
# El modelo TFLite debe soportar las operaciones básicas de TFLite así como
algunas de Tensorflow.
conversor_DRQ.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS,
    tf.lite.OpsSet.SELECT_TF_OPS
]
# Convertimos el modelo base con las especificaciones anteriores a un mode-
lo cuantizado con rango dinámico. Pesos cuantizados aquí ya que son "está-
ticos".
# En la función de evaluado y con el Interpreter es donde se cuantizan las
activaciones dinámicamente (toma max y min valor y lo acota a tipo de dato
adecuado), en tiempo de inferencia.
modelo_DRQ = conversor_DRQ.convert()

# Guardar el modelo en un archivo.
with open('modelo_DRQ.tflite', 'wb') as f:
    f.write(modelo_DRQ)
print("CUANTIZADO POSTENTRENAMIENTO CON RANGO DINÁMICO GUARDADO!")
```

A continuación, realizamos lo mismo para la cuantización con el tipo de datos FLOAT16. Se crea el objeto conversor, se eliminan otro tipo de optimizaciones, se establece el tipo de datos para las cuantizaciones, así como las operaciones que pueden realizar, para acabar convirtiendo el modelo. El código utilizado es el siguiente:

```
# <------------------------------FLOAT16 QUANTIZATION--------------------
---------->

# TFLite es Tensorflow para hardware más limitado, limita las funciones
que tenemos.
# Importamos el modelo base guardado.
conversor_F16Q = tf.lite.TFLiteConverter.from_keras_model(modelo)
# Dynamic Range Quantization no necesita un representative_dataset ya que
es Post Training Quantization, no Quantization Aware Training. Así que no
se lo pasamos.
# Aplicamos la optimización por defecto (la mejor que pueda) sin datos re-
presentativos. Se puede optimizar para mejor latencia/inferencia, tama-
ño...
conversor_F16Q.optimizations = [tf.lite.Optimize.DEFAULT]
# Establecemos los tipos de datos con el que cuantizar (pesos, activacio-
nes....), en este caso todo va a float16.
conversor_F16Q.target_spec.supported_types = [tf.float16]
```

```
# El modelo TFLite debe soportar las operaciones básicas de TFLite así
como algunas de Tensorflow.
conversor_F16Q.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS,
    tf.lite.OpsSet.SELECT_TF_OPS
]
# Convertimos el modelo base con las especificaciones anteriores a un mo-
delo cuantizado post-entrenamiento a float16. Pesos cuantizados aquí ya
que son "estáticos".
# En la función de evaluado y con el Interpreter es donde se cuantizan las
activaciones dinámicamente (toma max y min valor y lo acota a tipo de dato
adecuado), en tiempo de inferencia.
modelo_F16Q = conversor_F16Q.convert()

# Guardar el modelo en un archivo.
with open('modelo_F16Q.tflite', 'wb') as f:
    f.write(modelo_F16Q)
print("CUANTIZADO POSTENTRENAMIENTO CON FLOAT16 GUARDADO!")
```

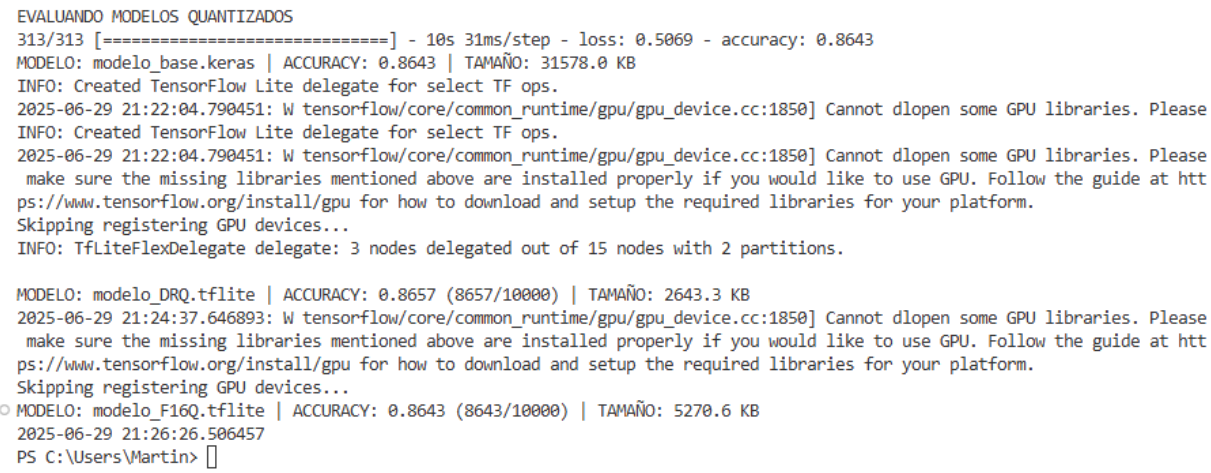
A partir de aquí se tienen los dos modelos cuantizados, con lo que se puede realizar la evaluación de los mismos, a través de un proceso de inferencia sobre los datos de test y la comprobación de cuántos aciertos ha obtenido el modelo. Para ello, se ha creado una función en Python para evaluar la cuantización. En ella, se crea un objeto intérprete que tomará el modelo cuantizado y generará las estructuras necesarias para iterar por los datos de test e inferir el resultado o la predicción, y a partir de ahí evaluar si la predicción equivale a la realidad.

El código que define esta función, así como la llamada a la misma para los dos modelos cuantizados, es el siguiente:

```
# <------------------------------EVALUACIÓN DE CUANTIZACIONES-------------
----------------->
# <------------------------------EVALUACIÓN DE CUANTIZACIONES-------------
----------------->


def evaluar_cuantizacion(rutaModelo, x_test, y_test):
    # Cargamos el modelo en el intérprete TFLite, capaz de realizar infe-
rencias.
    interpretador = tf.lite.Interpreter(model_path = rutaModelo)
    # Guardamos memoria para todas las estructuras que usa la red neuronal
(entrada, pesos, salida, activaciones...). Requerido para que el intérprete
sepa qué le va a llegar.
    interpretador.allocate_tensors()
```

```
    # Detalles para alimentar el modelo (estructura/forma, índice para
alimentar el modelo, datatype).
    detalle_entrada = interpretador.get_input_details()
    detalle_salida = interpretador.get_output_details()

    # Al convertirlo en algo iterable evitamos errores si nos llega otro ti-
po de estructura...
    y_test = np.array(y_test)
    # Contador de aciertos y de críticas totales.
    correctas = 0
    total = len(x_test)

    # Bucle para inferir sobre la muestra de test y comprobar cuanta Accura-
cy presenta.
    for i in range(total):
        # Lista con una sola review transformada en números, 170 números.
        entrada = x_test[i:i+1]
        # Requerido por el modelo TFLite.
        entrada = entrada.astype(np.float32)

        # Le damos al modelo la entrada procesada para que infiera.
        interpretador.set_tensor(detalle_entrada[0]['index'], entrada)
        interpretador.invoke()
        # Obtenemos la inferencia.
        salida = interpretador.get_tensor(detalle_salida[0]['index'])[0][0]

        # 0 es negativo, 1 es positivo
        prob = int(salida > 0.5)  # Clasificación binaria. np.argmax(salida)
para varias clases.
        if prob == y_test[i]:
            correctas += 1

    # Cálculo de precision e impresión de la misma y de tamaño de modelo a
nivel archivo.
    precision = correctas / total
    tamaño_modelo_kb = os.path.getsize(rutaModelo) / 1024
    print(f"MODELO: {os.path.basename(rutaModelo)} | ACCURACY: {preci-
sion:.4f} ({correctas}/{total}) | TAMAÑO: {tamaño_modelo_kb:.1f} KB")


print(f"EVALUANDO MODELOS QUANTIZADOS")
evaluacion = modelo.evaluate(np.asarray(X_test), np.asarray(y_test))
print(f"MODELO: {os.path.basename('modelo_base.keras')} | ACCURACY:
{round(evaluacion[1], 4)} | TAMAÑO: {os.path.getsize('modelo_base.keras') /
1024:.1f} KB")
evaluar_cuantizacion('modelo_DRQ.tflite', X_test, y_test)
evaluar_cuantizacion('modelo_F16Q.tflite', X_test, y_test)
```

La ejecución del siguiente código ha sido ciertamente rápida, principalmente debido a que el modelo original ya había sido guardado y entrenado. En total, la ejecución de esta parte de cuantización han sido unos seis minutos. La salida por pantalla no se puede exponer en su totalidad en este documento debido a ciertos mensajes de advertencia sobre algunas librerías para poder utilizar la GPU que no están instaladas, pero se puede observar la evaluación en la siguiente imagen:



Como podemos observar, el modelo base tiene una precisión del 86.43% con un tamaño de 31.5 megabytes aproximadamente, y los modelos cuantizados mejoran esto al igualar prácticamente la precisión (86.57% de precisión para la cuantización de rango dinámico, 86.43% de precisión por su parte para la cuantización en tipo de datos FLOAT16) y disminuyendo en varias magnitudes el tamaño del modelo (2.6 megabytes para la cuantización de rango dinámico, por su parte 5.2 megabytes para la cuantización con tipo de dato FLOAT16).

3.6 Podado.

Entrando ya en el siguiente script, llamado “4-Pruning.py”, lo primero que hacemos es ver las librerías necesarias para el script:

```
mport nltk
from nltk.corpus import stopwords
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
import tensorflow as tf
from tensorflow import keras

import tensorflow_model_optimization as tfmot
```

A continuación, se carga el modelo y se muestra la estructura de la red original, con el siguiente código:

```
print("CARGANDO MODELO")
modelo = keras.models.load_model('modelo_base.keras')
print("MODELO CARGADO CORRECTAMENTE")
print("RESUMEN ESTRUCTURA MODELO")
modelo.summary()
```

El resultado de ese código es el siguiente:

```
RESUMEN ESTRUCTURA MODELO
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 170, 128)          2560000

 lstm (LSTM)                 (None, 128)               131584

 dropout (Dropout)           (None, 128)               0

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 2,691,713
Trainable params: 2,691,713
Non-trainable params: 0
_________________________________________________________________
PS C:\Users\Martin>
```

La idea a continuación es definir ciertos parámetros requeridos para el modelo podado. El primero de ellos es el rango de porcentajes de pesos que se podarán durante el entrenamiento. Los valores comunes son el 20% de pesos podados desde el inicio (los que menos valor absoluto tengan), y el 80% de pesos podados al final del entrenamiento. Luego se definen las épocas o epochs que se harán como proceso de ajuste fino y reentrenamiento. Se define el paso desde el cual se empieza a realizar el podado, así como el paso en el cual finaliza, los pasos o steps es una actualización de los pesos del modelo, la cual ocurre tras procesar un batch de datos. En este caso lo hacemos desde el principio hasta el final de los epochs de reentrenamiento. También es importante hacer que en cada batch procesado se actualice el paso, de tal manera que el modelo sepa cuando aplicar el podado, de lo contrario no se podaría el modelo. Esto se hace a través de callbacks, que es la función que se llama en esta situación.

El código para estos pasos se muestra a continuación:

```
print("INICIO DE TECNICAS DE PRUNING")
# APLICAR PRUNING SOBRE MODELO YA ENTRENADO
print("APLICANDO PRUNING SOBRE EL MODELO ENTRENADO")

# Rango de proporción de pesos a cero al principio y final. Por lo general,
mejor entre 0.2 y 0.8.
# 25% de pesos podados inicialmente, no subirlo mucho ya que lo hacemos más
pequeño y menos preciso.
pct_inicio = 0.2
# 85% de pesos podados al final.
pct_final = 0.8
# Cuántos epochs reentrenamos el modelo con pruning. Es para que se adapte
el modelo cuando le has quitado pesos, no necesita muchos ya que ya está
entrenado.
pruning_epochs = 3
# Paso en el que comienza el podado. Aplicamos podado después de que haya
sido ya entrenado, en verdad estamos aplicándolo en los últimos 3 epochs.
begin_step = 0
# Pasos por epochs son cuanta muestra tienes entre cuantas muestras proce-
sas a la vez. Es un batch procesado, cada vez que pasa se actualiza el mo-
delo.
pasos_por_epochs = math.ceil(len(X_train) / batch)
# Paso en el que se para el podado, en este caso al final del último epoch
de fine-tuning. Si hubiesen más pasos, los pesos podados se quedan como es-
tán, el resto se actualizan.
end_step = pasos_por_epochs * pruning_epochs
# Cuando se entrene el modelo se actualiza el paso en cada batch para saber
cuándo aplicar o no la poda. Si no lo utilizamos no actualizará bien el
sparsity y el podado no avanza.
# Por cada batch se actualiza el paso y se calcula el nuevo sparsity.
callbacks = [tfmot.sparsity.keras.UpdatePruningStep()]
```

A continuación, ya se puede instanciar el objeto con el que se realizará el Podado basado en Magnitudes, la cual es de las formas más comunes de podado. A este objeto se le dispone del modelo base, así como una planificación de cómo y cuándo podar. Esta planificación se realiza disponiéndole al planificador del paso inicial y final y los porcentajes de poda inicial y final, y este aplicará la poda de manera progresiva y suave, aunque se puede modificar para que aplique mayor poda al principio o al final del reentrenamiento.

Cabe resaltar que, con esta implementación en Keras, la estructura del modelo no se ve modificada, esto significa que si, por ejemplo, utilizáramos la función summary() sobre el modelo podado, el resultado sería idéntico al resultado de la función summary() sobre el modelo original. Del mismo modo, si se guarda el modelo podado en un archivo, tendría algo menos de tamaño en disco (debido a que hemos perdido precisión en los pesos, pero no hemos eliminado neuronas del

modelo) que si se guarda el modelo original en un archivo. Esto es debido a que TFMOT (TensorFlow Model Optimizacion Toolkit, la librería requerida para hacer podado) lo que hace es aplicar una máscara sobre las neuronas y a esa máscara se le asigna un valor de cero.

Con todo esto, el modelo podado ya puede ser reentrenado. El código para estos últimos pasos mencionados es el siguiente:

```
# Necesario para aplicar Magnitude-Based Pruning de la libreria Tensorflow
Model Optimization Toolkit.
# Magnitude Based Pruning considera que los pesos cercanos a cero no influyen
lo suficiente en la salida del modelo como para que estén ahí aumentando la
complejidad del mismo.
# Por este motivo, este tipo de Pruning desactiva (pone a cero a través de
máscara) esos pesos con valores cercanos a cero y así reduce tamaño sin afec-
tar a la precisión.
prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

# Aplicar podado al modelo base, con función polinómica para que se vaya
aplicando con cierta progresión, versus el ConstantSparsity que lo aplica a
partir de cierto step.
modelo_podado = prune_low_magnitude(to_prune = modelo, pruning_schedule =
tfmot.sparsity.keras.PolynomialDecay(initial_sparsity = pct_inicio, fi-
nal_sparsity = pct_final, begin_step = 0, end_step = end_step))

# Recompilar el modelo podado.
modelo_podado.compile(optimizer = 'adam', loss = 'binary_crossentropy', me-
trics=['accuracy'])

# Reentrenar el modelo con podado.
print("REENTRENAMIENTO DEL MODELO PODADO")
modelo_podado.fit(X_train, y_train, batch_size = batch, epochs = prun-
ing_epochs, validation_split = 0.1, callbacks = callbacks)
```

Una vez el modelo está ya reentrenado con el mismo optimizador y función de pérdida que el modelo original, de tal manera que los resultados sean comparables. Se realiza la inferencia sobre los datos de test, y la evaluación para medir su precisión.

El código para este apartado es el siguiente:

```
# Comprobar el modelo con los datos de Test, asegurando que sean vectores.
# Obtenemos en una lista la pérdida y la accuracy, como se especificó en
Compile().
print("COMPROBAR MODELO CON DATOS DE TEST")
evaluacion = modelo.evaluate(np.asarray(X_test), np.asarray(y_test))
print(f"Loss: {round(evaluacion[0], 4)}, accuracy:
{round(evaluacion[1],2)*100}%")
evaluacion = modelo_podado.evaluate(np.asarray(X_test), np.asarray(y_test))
print(f"Loss: {round(evaluacion[0], 4)}, accuracy:
{round(evaluacion[1],2)*100}%")
```

El resultado de esta inferencia arroja los siguientes resultados:

```
REENTRENAMIENTO DEL MODELO PODADO
Epoch 1/3
563/563 [==============================] - 108s 186ms/step - loss: 0.0456 - accuracy: 0.9898 - val_loss: 0.4641 - val_accuracy: 0.8742
Epoch 2/3
563/563 [==============================] - 98s 173ms/step - loss: 0.0449 - accuracy: 0.9887 - val_loss: 0.4010 - val_accuracy: 0.8690
Epoch 3/3
563/563 [==============================] - 100s 177ms/step - loss: 0.0327 - accuracy: 0.9919 - val_loss: 0.5189 - val_accuracy: 0.8700
COMPROBAR MODELO CON DATOS DE TEST
313/313 [==============================] - 10s 29ms/step - loss: 0.5298 - accuracy: 0.8680
Loss: 0.5298, accuracy: 87.0%
313/313 [==============================] - 12s 36ms/step - loss: 0.5298 - accuracy: 0.8680
Loss: 0.5298, accuracy: 87.0%
PS C:\Users\Martin>
```

Por último, podemos tratar de ver si de verdad se ha aplicado el podado o no, con el objetivo de ver cuántos parámetros entrenables tiene el modelo podado. Para ello, podemos ver con la función id() que el modelo y el modelo podado son distintos, y eliminamos las estructuras del podado para poder contar por cada parámetro si el valor es cero (se ha podado) o no. El código para el siguiente apartado es el siguiente:

```
# Comprobar modelos distintos
print("ID del modelo base:", id(modelo))
print("ID del modelo podado:", id(modelo_podado))

# Calcular parámetros en los modelos.
print("CALCULANDO COMPARACIÓN DE PARÁMETROS EN MODELOS")
```

```python
# Función para contar los parámetros útiles y totales en el modelo podado, basado en el modelo base.
def contar_parametros(modelo):
    params_no_podados = 0
    params_totales = 0
    for capa in modelo.layers:
        for peso in capa.weights:
            # Si es un peso podado, en su nombre llevará mask, que indica que la máscara de podado está aplicada sobre el peso.
            if 'mask' not in peso.name:
                # Obtenemos el peso convertido en un array.
                valor = peso.numpy()
                # Todos los valores que tenga se cuentan como parámetros totales.
                params_totales += valor.size
                # Todos los valores que no sean ceros (si lo son entonces son podados) se tienen como parámetros útiles o no podados.
                params_no_podados += np.count_nonzero(valor)
    return params_no_podados, params_totales

# Contar parámetros para comparar podado.
params_utiles_podado, params_totales_base = contar_parametros(modelo_limpio_podado)
print(f"Parámetros útiles (no son cero) en modelo podado: {params_utiles_podado}")
print(f"Parámetros totales en modelo base: {params_totales_base}")
print(f"Proporción de podado (sparsity): {round(1 - params_utiles_podado / params_totales_base, 2) * 100}%.")

# Guardar modelo final podado, pesa menos que el original ya que hemos perdido detalle al pasar pesos cercanos a cero a cero.
print("GUARDANDO MODELO PODADO")
modelo_limpio_podado.save('modelo_podado.keras')
```

El resultado del código es el siguiente, y lo que podemos ver es que, en efecto, los modelos son distintos, por tanto, las coincidencias son reales, además de que se ha realizado un 80% de podado de pesos en el modelo.

```
ID del modelo base: 2345396841824
ID del modelo podado: 2345543085168
CALCULANDO COMPARACIÓN DE PARÁMETROS EN MODELOS
Parámetros útiles (no son cero) en modelo podado: 543847
Parámetros totales en modelo base: 2691713
Proporción de podado (sparsity): 80.0%.
GUARDANDO MODELO PODADO
```

3.7 Destilación de Conocimiento.

Para terminar este apartado de Diseño Metodológico e Implementación Práctica, se describe el último script, llamado “5-Knowledge Distillation.py”, lo primero que se realiza es la importación de librerías requeridas, así como cargar el que será el modelo profesor y establecer los pesos invariantes para este modelo profesor, ya que no se desea que las inferencias que haga el profesor modifiquen sus pesos, ya que de esta manera el modelo estudiante no terminaría de aprender.

Por otra parte, también se establecen los dos hiperparámetros requeridos para esta técnica de Destilación de Conocimiento, el valor Alpha y el valor de Temperatura. El primero de ellos indica con valores más cercanos a cero si el estudiante debe aprender más de la forma en la que infiere el profesor, o con valores más cercanos a uno si el estudiante debe trabajar más con los datos. El segundo de ellos es un valor necesario en la función softmax() requerida para la optimización, su valor por defecto es uno y estos valores pequeños hacen que dentro de la curva de probabilidades para cada respuesta resalte una respuesta con mucha mayor probabilidad que otras, mientras que si se aumenta la temperatura esta curva de probabilidades se aplana, haciendo que el modelo estudiante entienda que hay más respuestas posibles.

A la hora de definir el modelo estudiante se ha creado una función en Python en la que, simplemente, se mantiene la estructura del modelo profesor, pero se divide entre dos el tamaño de las capas, logrando así un modelo estudiante con la mitad de parámetros entrenables.

El código que implementa lo explicado en estos anteriores párrafos es el siguiente:

```
# Cargando modelo profesor.
print("CARGANDO MODELO")
modelo = keras.models.load_model('modelo_base.keras')
print("MODELO CARGADO CORRECTAMENTE")
modelo.trainable = False # Congelar pesos del modelo profesor

print("INICIO KNOWLEDGE DISTILLATION") # Hyperparams de Destilación
valor_Alpha = 0.5
valor_Temperatura = 1.25

# Crear modelo estudiante: misma estructura de capas que profesor, pero
reducido.
print("CREANDO MODELO ESTUDIANTE")
def crear_modelo_estudiante():
    modelo = keras.Sequential()
    modelo.add(Embedding(tamanio_diccionario, int(tamanio_capa / 2), in-
put_length = longitudMax))
    modelo.add(LSTM(int(tamanio_capa / 2)))
    modelo.add(Dropout(0.2))
    modelo.add(Dense(1, activation = 'sigmoid'))
    return modelo
estudiante = crear modelo estudiante()
```

Los valores elegidos para los hiperparámetros del modelo son 0.5 para Alpha y 1.25 para la Temperatura. Estos valores han sido escogidos debido a sus buenos resultados dentro de un extenso proceso de pruebas con otros valores de estos hiperparámetros. Durante alrededor de quince horas se probaron todos estos valores de Alpha (0.05, 0.15, 0.30, 0.50, 0.70, 0.85 y 0.95) contra todos estos valores de Temperatura (0.75, 1.00, 1.25, 1.50, 2.00, 3.00, 5.00). Teniendo en cuenta que el modelo profesor retorna precisiones de alrededor del 86.5%, las tres configuraciones que han arrojado mayores precisiones en el banco de pruebas automáticas han sido las siguientes:

1. alpha = 0.85, temperatura = 1.5 --> Loss = 0.2323, Accuracy = 0.8840.
2. alpha = 0.5, temperatura = 1.25 --> Loss = 0.0986, Accuracy = 0.8823.
3. alpha = 0.15, temperatura = 5.0 --> Loss = 0.4537, Accuracy = 0.8789.

Interpretando mínimamente estos resultados:

1. El estudiante del primer resultado aprende mayoritariamente de los datos (Alpha), y en la curva de probabilidades un resultado destaca sobre el resto (Temperatura).
2. El estudiante del segundo resultado es equilibrado, ya que balancea aprender del profesor y de los datos (Alpha) y en la curva de probabilidades un resultado destaca sobre el resto (Temperatura).
3. El estudiante del tercer resultado aprende bastante de cómo infiere el profesor (Alpha), y aplana bastante la curva de probabilidades, de tal forma que detecta que hay diferentes respuestas para la tarea a realizar (Temperatura).

Debido a este pequeño análisis de los resultados, se ha optado por trabajar con la primera configuración ya que se trata de primar la mayor precisión. Sin embargo, es perfectamente válido utilizar la segunda o la tercera configuración debido a que sus valores de precisión son muy similares. Además, aplicándolo a este proyecto la segunda configuración sería la más equilibrada y con mayor “sentido” en sus valores, ya que tiene un valor equilibrado de Alpha, y solo hay dos valores en la curva de probabilidades (crítica positiva o negativa), entonces se puede hacer que uno de los resultados sobresalga más que el otro, ya que es un problema binario.

El resultado del banco de pruebas automático realizado, tras alrededor de quince horas y 12 epochs para cada configuración, ordenado de mayor a menor precisión, es el siguiente:

- alpha = 0.85, temperatura = 1.5 --> Loss = 0.2323, Accuracy = 0.8840.
- alpha = 0.5, temperatura = 1.25 --> Loss = 0.0986, Accuracy = 0.8823.
- alpha = 0.15, temperatura = 5.0 --> Loss = 0.4537, Accuracy = 0.8789.

- alpha = 0.7, temperatura = 1.0 --> Loss = 0.1642, Accuracy = 0.8758.

- alpha = 0.7, temperatura = 1.0 --> Loss = 0.1642, Accuracy = 0.8758.
- alpha = 0.15, temperatura = 1.5 --> Loss = 0.2828, Accuracy = 0.8748.
- alpha = 0.85, temperatura = 3.0 --> Loss = 0.3330, Accuracy = 0.8737.
- alpha = 0.3, temperatura = 1.5 --> Loss = 0.2235, Accuracy = 0.8728.
- alpha = 0.15, temperatura = 1.0 --> Loss = 0.6445, Accuracy = 0.8700.
- alpha = 0.95, temperatura = 1.25 --> Loss = 0.1788, Accuracy = 0.8696.
- alpha = 0.3, temperatura = 3.0 --> Loss = 0.3343, Accuracy = 0.8691.
- alpha = 0.5, temperatura = 3.0 --> Loss = 0.6469, Accuracy = 0.8678.
- alpha = 0.7, temperatura = 2.0 --> Loss = 0.1473, Accuracy = 0.8678.
- alpha = 0.85, temperatura = 2.0 --> Loss = 0.2610, Accuracy = 0.8624.
- alpha = 0.05, temperatura = 0.75 --> Loss = 0.3363, Accuracy = 0.8616.
- alpha = 0.3, temperatura = 5.0 --> Loss = 0.2701, Accuracy = 0.8610.
- alpha = 0.95, temperatura = 2.0 --> Loss = 0.5474, Accuracy = 0.8609.
- alpha = 0.3, temperatura = 2.0 --> Loss = 0.5058, Accuracy = 0.8606.
- alpha = 0.5, temperatura = 1.5 --> Loss = 0.1725, Accuracy = 0.8592.
- alpha = 0.85, temperatura = 1.25 --> Loss = 0.2598, Accuracy = 0.8545.
- alpha = 0.05, temperatura = 1.0 --> Loss = 0.3358, Accuracy = 0.8521.
- alpha = 0.85, temperatura = 1.0 --> Loss = 0.2163, Accuracy = 0.8519.
- alpha = 0.3, temperatura = 1.0 --> Loss = 0.3881, Accuracy = 0.8492.
- alpha = 0.5, temperatura = 0.75 --> Loss = 0.3637, Accuracy = 0.8485.
- alpha = 0.15, temperatura = 2.0 --> Loss = 0.5343, Accuracy = 0.8480.
- alpha = 0.95, temperatura = 1.0 --> Loss = 0.3972, Accuracy = 0.8460.
- alpha = 0.7, temperatura = 1.25 --> Loss = 0.3119, Accuracy = 0.8455.
- alpha = 0.05, temperatura = 5.0 --> Loss = 0.2755, Accuracy = 0.8454.
- alpha = 0.05, temperatura = 3.0 --> Loss = 0.2046, Accuracy = 0.8447.
- alpha = 0.7, temperatura = 5.0 --> Loss = 0.3337, Accuracy = 0.8442.
- alpha = 0.5, temperatura = 5.0 --> Loss = 0.6770, Accuracy = 0.8436.
- alpha = 0.3, temperatura = 0.75 --> Loss = 0.3735, Accuracy = 0.8415.
- alpha = 0.15, temperatura = 3.0 --> Loss = 0.3898, Accuracy = 0.8400.
- alpha = 0.3, temperatura = 1.25 --> Loss = 0.3194, Accuracy = 0.8320.
- alpha = 0.7, temperatura = 3.0 --> Loss = 0.3794, Accuracy = 0.8216.
- alpha = 0.95, temperatura = 0.75 --> Loss = 0.2533, Accuracy = 0.8152.
- alpha = 0.85, temperatura = 0.75 --> Loss = 0.3073, Accuracy = 0.8120.

- alpha = 0.5, temperatura = 1.0 --> Loss = 0.4859, Accuracy = 0.8065.
- alpha = 0.05, temperatura = 2.0 --> Loss = 0.4344, Accuracy = 0.8042.
- alpha = 0.85, temperatura = 5.0 --> Loss = 0.3662, Accuracy = 0.8002.
- alpha = 0.7, temperatura = 0.75 --> Loss = 0.2802, Accuracy = 0.7905.
- alpha = 0.05, temperatura = 1.5 --> Loss = 0.3990, Accuracy = 0.7894.
- alpha = 0.05, temperatura = 1.25 --> Loss = 0.2949, Accuracy = 0.7848.
- alpha = 0.5, temperatura = 2.0 --> Loss = 0.3154, Accuracy = 0.7693.
- alpha = 0.95, temperatura = 5.0 --> Loss = 0.5802, Accuracy = 0.6507.
- alpha = 0.95, temperatura = 3.0 --> Loss = 0.6328, Accuracy = 0.5385.
- alpha = 0.15, temperatura = 1.25 --> Loss = 0.6089, Accuracy = 0.5351.
- alpha = 0.15, temperatura = 0.75 --> Loss = 0.6199, Accuracy = 0.5319.
- alpha = 0.95, temperatura = 1.5 --> Loss = 0.6887, Accuracy = 0.5025.

MEJOR CONFIGURACIÓN (12 epochs): alpha = 0.85, temperatura = 1.5 --> Accuracy = 0.8840.

Al probar la configuración con mayor precisión en solitario, vemos que el resultado no es el mismo, pero es muy similar. Cada ejecución puede dar una precisión distinta debido a la inicialización aleatoria de los pesos.

```
EVALUACION FINAL
Modelo Profesor:
313/313 [==============================] - 12s 36ms/step - loss: 0.5069 - accuracy: 0.8643

Modelo Estudiante:
313/313 [==============================] - 5s 17ms/step - binary_accuracy: 0.8762 - student_loss: 0.3533

Accuracy Profesor: 0.8643.
Accuracy Estudiante: 0.8762.
PS C:\Users\Martin>
```

Para implementar la Destilación de Conocimiento en Python se utilizará una clase, debido a que se necesita modificar o personalizar algunos comportamientos de algunas funciones importantes como fit() o compile(), utilizadas en los modelos de la librería Keras. El concepto de clase viene de la Programación Orientada a Objetos (POO), y es una forma de plantilla en donde definimos algunos valores intrínsecos (atributos) y algunos comportamientos (métodos/funciones) del objeto que queremos representar. Los objetos pueden heredar aspectos de una clase padre, y pueden sobrescribir funciones que hereden de esa clase padre. Toda clase tiene una función constructora, que se llama cuando se crea una instancia del objeto, y en la que normalmente se definen el valor de los atributos de la clase.

Se expone a continuación la clase definida para la destilación en su completitud para, posteriormente, ir explicando cada parte adecuadamente. El código completo de la clase es el siguiente:

```
print("DEFINIENDO CLASE DE DISTILACION")
class Distiller(Model):
    # Constructor de la clase, requerirá como atributos de la clase un mo-
delo maestro ya entreando y un modelo estudiante básico....
    def __init__(self, estudiante, maestro, alpha, temperatura):
        # Para que esta clase Distiller funcione, necesita el constructor
de su clase padre (Model) inicializado con sus diferentes funciones, por
eso lo llamamos.
        # Es como si declarasemos un objeto Model pero diciendo que "vamos
a definir un objeto personalizado", al inicializar ese objeto Model se po-
nen a disposición sus distintas funciones.
        # De esta manera, declaramos un objeto Modelo que se asigna a esta
clase Distiller, y que algunas de sus funciones las sobreescribiremos con
nuestras definiciones de las mismas.
        super(Distiller, self).__init__()
        # Asignamos los atributos maestro y estudiante que nos mandan a la
clase.
        self.maestro = maestro
        self.estudiante = estudiante
        # Asignamos hiperparámetros de la Destilación:
        # Asignar a Distiller el balance entre ponderar más Student Loss
(aprender de los datos, valor 1) o Distillation Loss (aprender de la manera
que trabaja el profesor, valor 0).
        self.alpha = alpha
        # Asignar a Distiller el valor de temperatura utilizados en la fun-
ción softmax() para los logits (raw output del modelo), valor por defecto
1, con menos una respuesta sobresaldrá mucho, con más aplanan la curva de
resultados.
        self.temperatura = temperatura

    # Sobreescribimos la funcion Model.compile() de Keras con la definición
de esta función, a la que le pasamos diferentes parámetros de la destila-
ción....
    def compile(self, optimizador, metrica, fnc_loss_rendimiento,
fnc_loss_destilacion):
        # Asigna la función compile() de la clase padre a Distiller, asig-
nando el optimizador y la metrica que hayan indicado, permite otros métodos
como fit() y evaluate()
        super(Distiller, self).compile(optimizer = optimizador, metrics =
metrica)
        # Asignar a Distiller el optimizador de rendimiento utilizado para
los gradientes y la retropropagación.
        self.optimizador = optimizador
        # Asignar a Distiller la función de pérdida para comparar predic-
ción del estudiante con la realidad.
        self.fnc_loss_rendimiento = fnc_loss_rendimiento
        # Asignar a Distiller la función de pérdida para comparar las pre-
dicciones del profesor con las del estudiante.
        self.fnc_loss_destilacion = fnc_loss_destilacion
```

```
    # Sobreescribimos la función de entrenamiento Model.train_step() de Keras
utilizada para los batches (todos los datos que pasan a la vez antes de que se
actualicen pesos).
    def train_step(self, batch_datos):
        # Divide el batch de datos en la entrada X y los resultados Y.
        x, y = batch_datos
        # Forward pass de la entrada al maestro sin que calcule gradientes, ya
que el modelo está ya entrenado.
        preds_maestro = self.maestro(x, training = False)

        # Manejamos el contexto de predicción con GradientTape(), el cual graba
las operaciones realizadas para poder recalcular los pesos más tarde.
        with tf.GradientTape() as tape:
            # Forward pass de la entrada al estudiante calculando gradientes en
base a su salida.
            preds_estudiante = self.estudiante(x, training = True)
            # Calcular función de pérdida entre predicción del estudiante y
realidad.
            loss_estudiante = self.fnc_loss_rendimiento(y, preds_estudiante)
            # Calcular función de pérdida de aprendizaje del profesor frente al
estudiante, utilizando la temperatura y función softmax() aplicado a las clases
(1, binaria en este caso), en vez de batches (0).
            loss_destilacion =
self.fnc_loss_destilacion(tf.nn.softmax(preds_maestro / self.temperatura,
axis=1), tf.nn.softmax(preds_estudiante / self.temperatura, axis = 1))
            # Calculamos la pérdida total combinando las dos pérdidas.
            loss_total = self.alpha * loss_estudiante + (1 - self.alpha) *
loss_destilacion

        # Obtenemos los parámetros entrenables del estudiante para calcular los
gradientes y actualizar estos parámetros.
        params_entrenables = self.estudiante.trainable_variables
        # Calculamos los gradientes teniendo en cuenta la pérdida y los paráme-
tros entrenables.
        gradientes = tape.gradient(loss_total, params_entrenables)
        # Aplicamos los gradientes a esos parámetros entrenables, actualizando y
haciendo que el modelo aprenda con la retropropagación.
        self.optimizador.apply_gradients(zip(gradientes, params_entrenables))
        # Actualizamos metricas según las prediciones realizadas, para saber el
rendimiento del modelo.
        self.compiled_metrics.update_state(y, preds_estudiante)
        # Guardamos en un diccionario todas las métricas para ir actualizándolas
según los epochs.
        metricas = {m.name: m.result() for m in self.metrics}
        # Añadimos a esos resultados la pérdida de rendimiento y de destilación.
        metricas.update({"student_loss": loss_estudiante, "distillation_loss":
loss_destilacion})
        # Devuelve los resultados de las métricas actualizadas.
        return metricas
```

```
    # Sobreescribimos la función de entrenamiento Model.test_step() de Keras
utilizada cuando se realiza la evaluación del modelo.
    def test_step(self, batch_data):
        # Divide el batch de datos en la entrada X y los resultados Y.
        x, y = batch_data
        # Predicción del estudiante sin que calcule gradientes, ya que el mo-
delo está ya entrenado.
        pred_estudiante = self.estudiante(x, training = False)
        # Calcular función de pérdida de precisión.
        loss_estudiante = self.fnc_loss_rendimiento(y, pred_estudiante)
        # Actualización de métricas del estudiantes en base a sus prediccio-
nes.
        self.compiled_metrics.update_state(y, pred_estudiante)
        # Guardamos las métricas del estudiante con sus valores actuales.
        metricas = {m.name: m.result() for m in self.metrics}
        # Actualizamos las métricas con su nuevo valor según predicciones
realizadas.
        metricas.update({"student_loss": loss_estudiante})
        # Devolvemos métricas actualizadas.
        return metricas
```

Se define una clase llamada Distiller, que heredará de la clase Model de la librería Keras, se trata de la librería estándar para creación de modelos de Inteligencia Artificial. La clase Distiller tendrá los atributos llamados estudiante, maestro, alpha y temperatura, con las que definiremos el modelo estudiante, maestro, y los valores de los hiperparámetros de la destilación; y tendrá funciones para sobrescribir las funciones compile(), train_step() y test_step() de su clase padre Model, así como la función __init__(), la cual es su función constructora.

La definición de la función constructora tiene como parámetros de entrada los valores que se asignarán en esta misma función a los atributos de la clase. Además de estas asignaciones, se realiza la herencia de la clase Model, lo que principalmente quiere decir que la clase Distiller recibe las funciones disponibles para la función Model.

Respecto a la función compile() la sobrescribiremos para esta clase Distiller dándole como parámetros de entrada el optimizador para el proceso de retropropagación, la métrica que observar (la precisión), la función de pérdida a minimizar para el rendimiento, así como la función de pérdida de destilación para reducir la distancia entre las predicciones del profesor y del estudiante.

Las funciones importantes que se van a sobrescribir son las funciones train_step() y test_step(), las cuales se utilizan durante los pasos o steps, ya sean durante el entrenamiento o la inferencia. Durante cada step se procesa un batch de datos, por tanto, la función de entrada de ambas funciones es un batch de datos, es decir, varias opiniones y su sentimiento que se procesan de manera paralela. Entrando ya

en las particularidades de la función train_step(), lo primero que se hace es separar las opiniones (x) de los sentimientos (y), y obtener la predicción que realiza el modelo profesor para ese batch de datos sin actualizar sus pesos. A continuación, se realiza la predicción del estudiante para ese batch de datos actualizando los pesos del modelo, y se utilizan las funciones de pérdida para calcular estas pérdidas de precisión y de destilación; luego de esto se calcula la pérdida total del modelo combinando las pérdidas de rendimiento y de destilación. A partir de esto, obtenemos los parámetros entrenables, se calculan los gradientes teniendo en cuenta la función de pérdida, para luego aplicar estos gradientes en un proceso de retropropagación que modificará los pesos y hará que el estudiante aprenda. Por último, en esta función se actualiza las métricas de precisión y pérdida combinada.

En la función test_step() en concreto, se divide el batch de datos en las reviews o críticas (x) y los sentimientos (y). Se realiza la predicción del estudiante sobre el batch de datos sin actualizar sus pesos (ya que el modelo se está evaluando), y se calcula la pérdida de rendimiento para actualizar y devolver las métricas actualizadas de precisión y de pérdida.

Una vez definida ya la clase con todos sus elementos, podemos generar un objeto de esta clase con el objetivo de entrenar y evaluar el modelo estudiante frente al modelo profesor. El modelo estudiante se realiza con la función de Python expuesta anteriormente, el modelo profesor es el modelo base original que se trata de optimizar, el valor de Alpha en 0.85 y el valor de Temperatura en 1.5. En cuanto a optimizadores para la retropropagación, función de pérdida de rendimiento, número de épocas y las métricas se utilizarán las mismas que el modelo profesor, para que sea comparable, y la función de pérdida de destilación utilizada será la función de divergencia de Kullback-Leibler (KL Divergence) por su buen funcionamiento con la función softmax().

Una vez el modelo estudiante está entrenado, se evalúan ambos con el mismo conjunto de datos de test y se imprime la precisión y la pérdida de ambos para su comparación. El código que implementa este comportamiento es el siguiente:

```
print("COMPILANDO Y ENTRENANDO DISTILLER")
# Crear clase Distiller con el estudiante y el profesor.
distiller = Distiller(estudiante, modelo, valor_Alpha, valor_Temperatura)
# Compilamos con mismas métricas que el profesor, para que sean compara-
bles.
distiller.compile(optimizador = keras.optimizers.Adam(), metrica = [me-
trics.BinaryAccuracy()], fnc_loss_destilacion = losses.KLDivergence(),
fnc_loss_rendimiento = losses.BinaryCrossentropy())
# Mostrar estructura del modelo estudiante.
print(estudiante.summary())
# Imprimir configuración de hiperparámetros.
print(f"\nEntrenando con alpha = {valor_Alpha}, temperatura = {va-
lor_Temperatura}.")
```

```
 Entrenamiento del modelo distilado, se guarda en variable para ver más
adelante las métricas en cada momento.
history = distiller.fit(X_train, y_train, epochs = epochs, batch_size =
batch, validation_data = (X_test, y_test))


# Evaluaciones del modelo estudiante y profesor.
print("\nEVALUACION FINAL")
print("Modelo Profesor:")
teacher_eval = modelo.evaluate(X_test, y_test)

print("\nModelo Estudiante:")
student_eval = distiller.evaluate(X_test, y_test)

print(f"\nAccuracy Profesor: {teacher_eval[1]:.4f}.")
print(f"Accuracy Estudiante: {student_eval[0]:.4f}.")

# Guardar el modelo destilado.
print("GUARDANDO MODELO")
distiller.save('modelo_destilado.keras')
print("MODELO GUARDADO CORRECTAMENTE")
```

Con esta ejecución acaba el apartado práctico del proyecto. A continuación, se resumirán los resultados vistos en las ejecuciones de los códigos y se obtendrán conclusiones de los mismos.

4. Resultados, conclusiones y líneas futuras.

En este apartado se realizará un compendio de los resultados obtenidos, así como unas conclusiones sobre el proyecto en su totalidad y posibles líneas futuras de trabajo que hayan quedado abiertas en este proyecto.

4.1 Resumen de resultados.

A continuación, se muestra una tabla resumen de los resultados obtenidos para los diferentes modelos expuestos durante la parte práctica:

Optimización	Precisión max en Test	Tamaño en disco	Parámetros entrenables	Tiempo de entrenamiento	Complejidad de implementación
Sin optimización	86.43%	31578 Kilobytes	Aprox. 2.7 millones	30 minutos	Base
Dynamic Range Quantization	86.57%	2644 Kilobytes	Aprox. 2.7 millones	6-8 minutos	Media
FLOAT16 Quantization	86.43%	5271 Kilobytes	Aprox. 2.7 millones	4-6 minutos	Media
Magnitude Based Pruning	86.4%	10535 Kilobytes	80% del total, 543847 parámetros	12-15 minutos	Sencilla
Response-Based Offline Knowledge Distillation	88.4%	5149 Kilobytes	50% del total, 1.31 millones	25 minutos	Alta

Como vemos, los modelos presentan una precisión muy similar entre ellos, siendo esta de alrededor del 87%. Además de esto, vemos diferencias notables entre el modelo base y los modelos optimizados en cuanto a tamaño en disco y tiempo de entrenamiento, teniendo los modelos optimizados mejores valores que el modelo base. Por último, a nivel complejidad de implementación y dentro de los modelos optimizados, se han tenido algunos pocos problemas más a la hora de desarrollar los códigos para Cuantización (por cómo es el modelo base y por las operaciones extendidas de la Cuantización) que para el código de Podado o el código de Destilación de Conocimiento.

4.2 Conclusiones.

Con los resultados descritos previamente, se puede proceder a sacar ciertas conclusiones sobre ellos. La primera conclusión obtenida según los resultados es que estas técnicas mantienen gran parte del rendimiento o precisión obtenido por el modelo base. Se podría pensar que al optimizar el modelo la precisión sea menor que la provista por el modelo base. Sin embargo, con las pruebas realizadas podemos concluir que, con un buen ajuste, la pérdida de rendimiento es marginal; de hecho, incluso se puede sobrepasar la precisión del modelo base. Cabe destacar que, según lo observado, estas optimizaciones no van enfocadas a ganar varios puntos porcentuales de precisión, es poco común que con estas técnicas se pase de un 86% a, por ejemplo, un 95% de precisión; su optimización viene por otra parte que a continuación se expone.

La segunda conclusión obtenida según los resultados es que estas técnicas de optimización parecen estar más enfocadas a reducir el tamaño de los modelos sin que estos pierdan precisión, más que a obtener un mayor rendimiento sobre el mismo modelo base. Como hemos visto, todos los modelos optimizados son bastante más ligeros en cuanto a tamaño en disco que el modelo base, se observan reducciones de tamaño de entre el 60% y el 90% del tamaño del modelo base, siendo la Cuantización la técnica que más reduce este tamaño. Como sabemos, la cuantización reduce el tamaño del modelo al utilizar tipos de datos menos definidos, y el podado reduce el tamaño al desactivar neuronas que aportan poco al resultado del modelo; por tanto, para dispositivos con hardware más limitado, estas técnicas son idóneas ya que se obtiene el mismo rendimiento adelgazando el tamaño en disco.

La tercera conclusión obtenida según los resultados es que estas técnicas de optimización funcionan muy bien para realizar el proceso de inferencia de una manera más rápida, se tendrá que entrenar el modelo base para así poder guardarlo y de ahí importarlo y poder optimizarlo, pero, al tener modelos más pequeños, la tarea de ajuste fino e inferencia será más eficiente en comparación con su modelo base. Esto, de vuelta y como se ha visto en la segunda conclusión, vuelve a favorecer a los dispositivos con menos recursos o potencia, ya que les permite operar con modelos grandes pero optimizados a sus capacidades.

La cuarta conclusión obtenida a través de las pruebas realizadas es que, para este proyecto y este caso de análisis de sentimientos sobre opiniones en IMDB Reviews, la técnica de cuantización sobresale por delante del resto en cuanto a reducción de tamaño, pero la técnica de Destilación de Conocimiento destaca por la precisión obtenida, mejorando la precisión del modelo base. La Cuantización puede reducir el tamaño del modelo en un 90% y aumentar mínimamente la precisión del mismo, y en caso de no querer tanta reducción, se puede mantener el detalle y precisión del modelo reduciendo su tamaño en más de un 60%. La técnica de Podado no ha brillado en este apartado debido a que la librería TFMOT puede podar capas densas y LSTMs, pero no puede podar la capa de Embedding, la cual en nuestro caso es la que contiene la gran mayoría de parámetros entrenables. La capa Embedding no la

puede podar debido a que no está basada en operaciones de matrices de la misma manera que lo están las capas Densas. La técnica de Destilación de Conocimiento ha conseguido aumentar la precisión del modelo, aunque el proceso de entrenamiento lleva prácticamente el mismo tiempo,

La quinta y última conclusión obtenida es que la optimización debe ser una parte fundamental del desarrollo de modelos, aunque no se vaya a desplegar el modelo en dispositivos limitados. Por poner una situación hipotética, si se crea un modelo original que utilice el 100% de las capacidades del dispositivo final y mejorándolo se obtiene un modelo optimizado que utiliza el 80% de las capacidades del dispositivo final, se podría utilizar otro dispositivo secundario más potente para crear un modelo que utilice el 120% de las capacidades del dispositivo final para después mejorarlo y obtener un modelo optimizado de mayor complejidad y potencia, y que utilice el 100% de las capacidades del dispositivo final. Cabe destacar que esta relación no es directa y entran en juego la tarea para la que esté diseñada el modelo, así como otra gran cantidad de factores.

4.3 Líneas de trabajo futuro.

Por la parte de las líneas y trabajo futuro, este trabajo ha sido detallado en cuanto a la descripción teórica y práctica de cuantización, podado y destilación de conocimiento, así que una línea de trabajo futuro puede ir tanto por explorar más formas prácticas de implementación de las técnicas de optimización expuestas, como ampliar el set de técnicas investigadas. En la parte teórica se han explicado varias formas de implementación dentro de cada técnica, se podrían implementar de manera práctica estas diferentes formas y compararlas, y también se podría profundizar en otras técnicas de optimización, como Low Rank Adaptation (LoRA).

Por último, una línea de trabajo futuro muy interesante pasa por la combinación de estas técnicas. Hay diferentes maneras de combinar las técnicas, un ejemplo podría ser obtener un modelo base de grandes capacidades para después podarlo, y a ese modelo podado se le podría destilar el conocimiento hacia un modelo de menor tamaño que, finalmente, se cuantice con el objetivo de que pierda una gran parte de su tamaño manteniendo su rendimiento, de tal forma que se pueda ejecutar en dispositivos con menos recursos. Salvo pequeños problemas de compatibilidad que puedan existir entre técnicas de optimización (debido a cómo son y cómo funcionan por dentro), nada impide combinar estas técnicas y aumentar el área de exploración del proyecto.

5. Referencias y webgrafía.

Este proyecto ha sido posible gracias a numerosas personas que, a través de Internet o de manera presencial, han dejado su grano de conocimiento y experiencia para que otras personas como yo puedan aprender de la materia e investigar por su cuenta para desarrollar proyectos.

A continuación, se muestra un listado de las fuentes utilizadas para este proyecto:

- Apuntes de la asignatura de Técnicas Analíticas del Máster en Analítica de Negocio y Big Data de la Universidad de Alcalá, del año 2024.
- Kaggle (s.f.) IMDb dataset of 50 K movie reviews. Disponible en: https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews (Accedido: 19 marzo 2025)
- IBM (s.f.) *Natural Language Processing*. Disponible en: https://www.ibm.com/think/topics/natural-language-processing (Accedido: 19 marzo 2025).
- DL.AI (s.f.) Natural Language Processing. Disponible en: https://www.deeplearning.ai/resources/natural-language-processing (Accedido: 19 marzo 2025)
- IBM (s.f.) Tokens. Disponible en: https://www.ibm.com/docs/en/watsonx/saas?topic=solutions-tokens (Accedido: 20 marzo 2025)
- IBM (s.f.) Recurrent Neural Networks. Disponible en: https://www.ibm.com/think/topics/recurrent-neural-networks (Accedido: 21 marzo 2025)
- Olah, C. (2015) Understanding LSTMs. Disponible en: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (Accedido: 22 marzo 2025)
- GeeksforGeeks (s.f.) Mathematical understanding of RNN and its variants. Disponible en: https://www.geeksforgeeks.org/mathematical-understanding-of-rnn-and-its-variants/ (Accedido: 23 marzo 2025)
- Krgngr, L. (s.f.) RNN and LSTM: simple mathematical explanations. Disponible en: https://medium.com/@lkrgngr/rnn-and-lstm-simple-mathematical-explanations-54167e04227b (Accedido: 24 marzo 2025)
- YouTube (s.f.) Vídeo RNN/LSTM explicación [vídeo]. Disponible en: https://www.youtube.com/watch?v=Keqep_PKrY8 (Accedido: 25 marzo 2025)
- Olah, C. (2014) NLP & RNNs Representations. Disponible en: https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/ (Accedido: 26 marzo 2025)

- Data-Science (s.f.) Illustrated guide to LSTMs and GRU's. Disponible en: https://medium.com/data-science/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 (Accedido: 27 marzo 2025)
- Britz, D. (s.f.) Recurrent neural networks tutorial – part 3. Disponible en: https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-3/ (Accedido: 28 marzo 2025)
- Britz, D. (s.f.) Recurrent neural networks tutorial – part 4. Disponible en: https://dennybritz.com/posts/wildml/recurrent-neural-networks-tutorial-part-4/ (Accedido: 29 marzo 2025)
- UVADLC (2016) Lecture 8 (PDF). Disponible en: https://uvadlc.github.io/lectures/nov2016/lecture8.pdf (Accedido: 30 marzo 2025)
- YouTube (s.f.) RNN/GRU playlist [lista de reproducción]. Disponible en: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi (Accedido: 31 marzo 2025)
- Amanatulla, A. (s.f.) Transformer architecture explained. Disponible en: https://medium.com/@amanatulla1606/transformer-architecture-explained-2c49e2257b4c (Accedido: 1 abril 2025)
- Kerem, A. (s.f.) Everything to learn about large language models – part 1/3. Disponible en: https://medium.com/@aydinKerem/everything-to-learn-about-large-language-models-part-1-3-how-do-llms-work-789e3c946282 (Accedido: 2 abril 2025)
- Data-Science at Microsoft (s.f.) How large language models work. Disponible en: https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f (Accedido: 3 abril 2025)
- Singh, V. (s.f.) LLM architectures explained. Disponible en: https://medium.com/@vipra_singh/llm-architectures-explained-nlp-fundamentals-part-1-de5bf75e553a (Accedido: 4 abril 2025)
- Geekflare (s.f.) Natural language understanding. Disponible en: https://geekflare.com/ai/natural-language-understanding/ (Accedido: 5 abril 2025)
- Vaswani, A. et al. (2017) Attention is all you need. Disponible en: https://arxiv.org/pdf/1706.03762 (Accedido: 6 abril 2025)
- DL.AI (s.f.) Quantization fundamentals with Hugging Face. Disponible en: https://www.deeplearning.ai/short-courses/quantization-fundamentals-with-hugging-face/ (Accedido: 5 mayo 2025)
- DL.AI (s.f.) Quantization in depth. Disponible en: https://www.deeplearning.ai/short-courses/quantization-in-depth/ (Accedido: 8 mayo 2025)

- Sachinsoni (s.f.) Introduction to model quantization. Disponible en: https://medium.com/@sachinsoni600517/introduction-to-model-quantization-4effc7a17000 (Accedido: 9 mayo 2025)
- AI Bites (s.f.) Model quantization in deep neural networks. Disponible en: https://medium.com/@AIBites/model-quantization-in-deep-neural-networks-81df49f3c7d8 (Accedido: 9 mayo 2025)
- Data-Science (s.f.) Quantizing neural network models. Disponible en: https://medium.com/data-science/quantizing-neural-network-models-8ce49332f1d3 (Accedido: 10 mayo 2025)
- Kezmann, J. (s.f.) Master the art of quantization. Disponible en: https://medium.com/@jan_marcel_kezmann/master-the-art-of-quantization-a-practical-guide-e74d7aad24f9 (Accedido: 10 mayo 2025)
- Evan W. (s.f.) Float toy. Disponible en: https://evanw.github.io/float-toy/ (Accedido: 11 mayo 2025)
- Vishalindev (s.f.) Understanding FP32, FP16 and INT8 precision. Disponible en: https://medium.com/@vishalindev/understanding-fp32-fp16-and-int8-precision-in-deep-learning-models-why-int8-calibration-is-5406b1c815a8 (Accedido: 11 mayo 2025)
- CodersCorner (s.f.) Floating-point representation. Disponible en: https://medium.com/coderscorner/floating-point-representation-63114653c9ee (Accedido: 11 mayo 2025)
- Jasminewu (s.f.) BF16, FP16, INT8 in LLM. Disponible en: https://medium.com/@jasminewu_yi/bf16-fp16-int8-in-llm-387912b41e45 (Accedido: 17 mayo 2025)
- YouTube (s.f.) Vídeo cuantización FP explanation [vídeo]. Disponible en: https://www.youtube.com/watch?v=UQlsqdwCQdc (Accedido: 17 mayo 2025)
- Kezmann, J. M. (s.f.) Master the art of quantization. Disponible en: https://medium.com/data-science/quantizing-neural-network-models-8ce49332f1d3 (Accedido: 18 mayo 2025)
- Kezmann, J. M. (s.f.) Master the art of quantization: practical guide (enlace repetido). Disponible en: https://medium.com/@jan_marcel_kezmann/master-the-art-of-quantization-a-practical-guide-e74d7aad24f9 (Accedido: 18 mayo 2025)
- Anhtuan (s.f.) Introduction to pruning. Disponible en: https://medium.com/@anhtuan_40207/introduction-to-pruning-4d60ea4e81e9 (Accedido: 24 mayo 2025)
- Harisudhan (s.f.) Pruning and distillation for accelerating LLM inference. Disponible en: https://medium.com/@speaktoharisudhan/pruning-and-

distillation-for-accelerating-llm-inference-754373d0f053 (Accedido: 24 mayo 2025)

- Kezmann, J. M. (s.f.) Optimizing deep learning models with pruning. Disponible en: https://medium.com/@jan_marcel_kezmann/optimizing-deep-learning-models-with-pruning-a-practical-guide-163e990c02af (Accedido: 25 mayo 2025)
- Siddiqui, M. (s.f.) Difference between structured and unstructured pruning. Disponible en: https://medium.com/@mhammadsiddiqui/difference-between-structured-and-unstructured-pruning-in-neural-cca5603581fb (Accedido: 25 mayo 2025)
- IBM (s.f.) Knowledge distillation. Disponible en: https://www.ibm.com/think/topics/knowledge-distillation (Accedido: 6 julio 2025)
- HuggingFace (s.f.) Knowledge distillation blog. Disponible en: https://huggingface.co/blog/Kseniase/kd (Accedido: 6 julio 2025)
- Neptune.ai (s.f.) Deep learning model optimization methods. Disponible en: https://neptune.ai/blog/deep-learning-model-optimization-methods (Accedido: 6 julio 2025)
- IBM (s.f.) LoRA. Disponible en: https://www.ibm.com/think/topics/lora (Accedido: 12 julio 2025)
- LoRA et al. (2020) LoRA: Low-Rank Adaptation. Disponible en: https://arxiv.org/abs/2006.05525 (Accedido: 12 julio 2025)
- TypingMind (s.f.) OpenAI O3 Mini vs DeepSeek R1. Disponible en: https://blog.typingmind.com/openai-o3-mini-vs-deepseek-r1/ (Accedido: 12 julio 2025)
- Xiao Bi (2024) DeepSeek LLM Scaling Open-Source Language Models with Longtermism. Disponible en: https://arxiv.org/pdf/2401.02954 (Accedido: 13 julio 2025)
- Vaswani, A. et al. (2015) Efficient estimation of word representations. Disponible en: https://arxiv.org/pdf/1503.02531 (Accedido: 13 julio 2025)
- Satya (s.f.) Understanding knowledge distillation in simple steps. Disponible en: https://medium.com/@satya15july_11937/understanding-knowledge-distillation-in-simple-steps-3310ef01f5e0 (Accedido: 19 julio 2025)
- Jenrola O. (s.f.) Exploring knowledge distillation in LLMs. Disponible en: https://medium.com/@jenrola_odun/exploring-knowledge-distillation-in-large-language-models-9d9be2bff669 (Accedido: 19 julio 2025)

- DS StackExchange (s.f.) Number of parameters in an LSTM model. Disponible en: https://datascience.stackexchange.com/questions/10615/number-of-parameters-in-an-lstm-model (Accedido: 2 julio 2025)
- StackOverflow (s.f.) Keras Dense layer output shape. Disponible en: https://stackoverflow.com/questions/61560888/keras-dense-layer-output-shape (Accedido: 2 julio 2025)
- StackOverflow (s.f.) Meaning of 'None' in model summary of Keras. Disponible en: https://stackoverflow.com/questions/47240348/what-is-the-meaning-of-the-none-in-model-summary-of-keras (Accedido: 4 julio 2025)
- Keras (s.f.) Better knowledge distillation recipe. Disponible en: https://keras.io/examples/keras_recipes/better_knowledge_distillation/ (Accedido: 6 julio 2025)
- Keras (s.f.) Vision knowledge distillation. Disponible en: https://keras.io/examples/vision/knowledge_distillation/ (Accedido: 6 julio 2025)
- TensorFlow (s.f.) tf.GradientTape API docs. Disponible en: https://www.tensorflow.org/api_docs/python/tf/GradientTape (Accedido: 7 julio 2025)