LAS MATEMÁTICAS DE LA ATENCIÓN

Lucía Sánchez Arrabal

LAS MATEMÁTICAS DE LA ATENCIÓN

Lucía Sánchez Arrabal

Memoria presentada como parte de los requisitos para la obtención del título de Grado en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Miguel Ángel Gutiérrez Naranjo

Índice general

English Abstract

Nowadays, Artificial Intelligence covers an increasingly number of areas both in the scientific field and in everyday life, such as automatic driving, image classification, facial recognition, etc.

One of the fields where Artificial Intelligence is contributing the most is at natural language processing, whose objective is to replicate the typical language of humans. In this report the mathematical concepts involved in these models will be explained. The attention mechanism will be studied. Apart from other tasks, it makes possible the operation of the Transformer, the neural network which is behind text generation models and which has been a point of reference to carry out others tasks.

However, before looking into the attention mechanism, an introduction to Artificial Intelligence is necessary, specifically to Machine Learning and Deep Learning, studying the mathematical aspects behind these fields. Therefore, the necessary knowledge to apply the attention mechanism to Deep Learning models will be obtained.

Finally, it will be shown that the theoretical model developed in the study is able to achieve a good outcome. For that purpose, a Python example is presented using the package Keras. The model will be trained with data containing English sentences along with its corresponding Spanish translations.

Resumen

En la actualidad, la Inteligencia Artificial abarca cada vez más ámbitos tanto en el campo científico como en la vida cotidiana, tales como conducción automática, clasificación de imágenes, reconocimiento facial, etc.

Uno de los campos en los que la Inteligencia Artificial está aportando grandes avances es en el procesamiento del lenguaje natural, cuya función es replicar el lenguaje natural propio de los humanos. En el presente trabajo vamos a ver los conceptos matemáticos implicados en estos modelos. Se estudiará el mecanismo de atención, que además de utilizarse en otros ámbitos, hace posible el funcionamiento de los Transformers, la red neuronal que hay detrás de los modelos generadores de texto y que ha servido como referente a la hora de realizar distintas tareas.

No obstante, antes de adentrarnos en el mecanismo de atención, es necesario hacer una introducción a la Inteligencia Artificial; más concretamente al *Machine Learning* y al *Deep Learning*, estudiando los aspectos matemáticos que son la base de estas disciplinas. De esta forma, tendremos el conocimiento necesario para poder aplicar el mecanismo de atención en modelos propios del *Deep Learning*.

Para terminar, se mostrará que el modelo teórico desarrollado en el trabajo obtiene buenos resultados. Para ello se expone un ejemplo con código Python usando la librería Keras. El modelo se entrenará a través de datos que contienen oraciones en inglés junto con su traducción al español.

1 Introducción a la Inteligencia Artificial

1.1 Inteligencia Artificial

La Inteligencia Artificial (IA) es una rama de las Ciencias de la Computación que nace en la década de 1950 con la pregunta de si los ordenadores pueden pensar o actuar de forma parecida a cómo los humanos lo hacen. El término *Inteligencia Artificial*, fue acuñado en 1956, cuando John McCarthy organizó una conferencia en Darmouth con el objetivo de responder a la pregunta planteada [3, Pág 2].

Dar una descripción formal de qué es la Inteligencia Artificial es una tarea difícil, aunque podemos dar diferentes definiciones según autores que destacan en la materia.

En [10], John McCarthy la define como la ciencia ó ingeniería de hacer máquinas inteligentes, especialmente programas inteligentes. Pero, ¿a qué nos referimos con "inteligencia"? Pues bien, la inteligencia según McCarthy es la parte computacional de la habilidad para resolver problemas y objetivos. Además existen distintos grados de inteligencia. Es por ello por lo que es difícil saber que procedimientos computacionales son inteligentes y cuales no lo son.

Una definición quizás más fácil de entender la da Chollet en [3, Sección 1.1.1]: "Es el esfuerzo de automatizar tareas que normalmente son realizadas por humanos o requieren de la inteligencia humana ".

Muchas más definiciones de IA se han dado a lo largo de la historia. En [16], las definiciones son clasificadas según 4 criterios, comparando la Inteligencia Artificial con el pensamiento humano, el comportamiento humano, el pensamiento racional y el comportamiento racional. Por ejemplo, veamos dos definiciones dadas en el libro

citado:

> "*The study of how to make computers do things at which, at the moment, people are better.*" (Rich and Knight, 1991)"

> "*Computational Intelligence is the study of the design of intelligent agents*" (Poole et al., 1998)

Ambas definiciones se basan en diseñar o hacer máquinas inteligentes. En el primer caso la inteligencia se mide respecto de la inteligencia humana. En el segundo caso se habla de forma más general sobre la inteligencia. La primera definición se clasifica en el grupo de comportamiento humano, y la segunda, en el grupo de comportamiento racional.

Un programa que sea capaz de jugar al ajedrez es un ejemplo de Inteligencia Artificial, pero no es más que una serie de reglas escritas por un programador en cierto lenguaje de programación. Este tipo de IA se conoce como "simbólica". Es la que se desarrolló entre los años 50 y 80, y puede resolver problemas de lógica. Sin embargo no es capaz de resolver problemas como la clasificación de imágenes o el procesamiento del lenguaje natural [3, Pág 3].

Dentro de la Inteligencia Artificial se encuentra el *Machine Learning* y este a su vez engloba al *Deep Learning*. Estos términos resultan familiares y se estudiarán con cierto detalle a lo largo del trabajo. En el siguiente gráfico podemos ver una relación entre los campos citados:

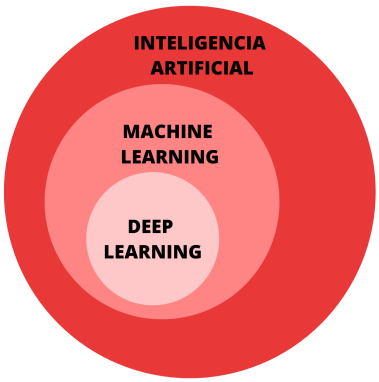


Figura 1.1: Relación entre Inteligencia Artificial, *Machine Learning* y *Deep Learning* .

Los principios de la Inteligencia Artificial no están claros, pues ya en el siglo XIX Charles Babbage inventó la máquina analítica, una máquina cuyo propósito era automatizar los cálculos realizados por personas. Se considera el primer ordenador del mundo. Ada Lovelace, matemática londinense, escribió unas notas en 1843 sobre este ordenador donde fue más allá del propósito de la máquina [3, Pág 3].

> "*La máquina analítica no tiene pretensión cualquiera de originar algo. Puede hacer cualquier cosa que sepamos ordenarle. Puede seguir el análisis, pero no es capaz de anticipar relaciones ni verdades analíticas...*" Ada Lovelace, Notes. 1843.

Esta nota fue citada por Alan Turing en un artículo de 1950, [21], donde introdujo un test para probar si una máquina era inteligente o no. El test consiste en qué un hombre, el interrogador, mantenga una conversación con una IA y con otro humano, sin saber quién es quién. Si el interrogador no es capaz de distinguir la máquina del humano diremos que la máquina ha superado el test y es inteligente. Turing reemplaza la pregunta de *"¿Las máquinas pueden pensar?"* por *"¿El interrogador puede distinguir a la IA del humano?"*.

Para que un programa pueda pasar este test se necesitan habilidades tales como el procesamiento del lenguaje natural, para poder tener una conversación con el interrogador; representación del conocimiento, para almacenar lo aprendido; razonamiento automático, para poder usar el conocimiento adquirido; y aprendizaje automático, para poder improvisar en circunstancias poco habituales para la máquina [16, Cap. 1].

1.2 *Machine Learning*

La Inteligencia Artificial simbólica necesita de un programador que escriba una serie de reglas en el ordenador para que este ejecute cierta tarea sobre ciertos datos. La diferencia en el *Machine Learning* es que el ordenador no va a necesitar seguir un programa para realizar un trabajo, si no que a través de unos datos y una salida esperada ligadas a esos datos y a la tarea que se quiera realizar, el ordenador debe aprender las reglas que debe usar para realizar la tarea cuando tome como entrada datos nuevos que nunca haya visto. Para inferir estas reglas el ordenador debe realizar un proceso de entrenamiento. Este entrenamiento necesita de técnicas estadísticas que mejoren en cada paso el resultado obtenido [3, Pág 4].

1.2.1 Definición y conceptos básicos

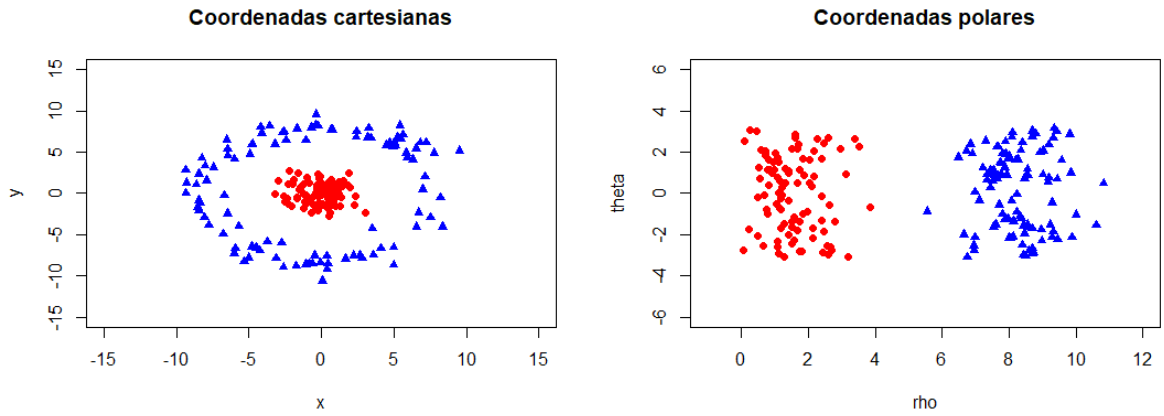
El *Machine Learning* es una rama de la Inteligencia Artificial. Un algoritmo de este campo toma como entrada un conjunto de datos formado por pares (x, y). x hace referencia a la entrada e y a la salida esperada. El algoritmo también necesita una función que mida el error entre los valores esperados de salida (que son datos de entrada) y los valores que el algoritmo da como salida realmente.

Veamos un ejemplo para mayor claridad: Si queremos clasificar imágenes de perros y gatos, el primer conjunto de datos será un conjunto de imágenes y el segundo será una etiqueta que diga si en cierta imagen hay un gato o un perro. A través de estos datos queremos que nuestro algoritmo sea capaz de distinguir en una nueva imagen un gato de un perro.

Por tanto, el objetivo es que la máquina obtenga información relevante sobre los datos. Para ello, tiene una gran influencia la representación de estos. Una tarea habitual en Aprendizaje Automático es la de clasificación o regresión. La diferencia entre ambas es que en la tarea de clasificación, las etiquetas de los valores de entrada son discretas; en cambio, las etiquetas asociadas a los datos de entrada en una tarea de regresión son continuas.

Supongamos que tenemos dos tipos de datos y los queremos clasificar en 2 grupos. Los conjuntos de datos forman dos coronas circulares cuando se representan en coordenadas cartesianas. Si los queremos separar por una línea recta la tarea se vuelve imposible. Sin embargo, si representamos el mismo conjunto de datos en coordenadas polares el problema se resuelve de forma sencilla, como podemos ver en la figura 1.2 [4].

Además de aprender que características debemos extraer de nuestro conjunto de datos, es importante también aprender que representación es la conveniente para aplicar cierto algoritmo de *Machine Learning* . Estas representaciones no son inventadas por la máquina, sino que buscan las posibles soluciones en un espacio de hipótesis. Imaginemos que tenemos varios puntos en $\mathbb{R}^2$ de dos colores distintos, blanco y negro. Queremos saber, dado un punto nuevo, de qué color será. Para ello, una posible solución es representar los puntos de manera que los podamos separar unos de otros. Dependiendo de la posición inicial de los puntos, puede ser que solo necesitemos rotarlos, desplazarlos o aplicar alguna transformación lineal. El conjunto de todos los cambios de coordenadas que se pueden hacer en dimensión 2 es el espacio de hipótesis en este caso [3, Págs 6-7].



(a) Coordenadas cartesianas.

(b) Coordenadas polares.

Figura 1.2: Representación de datos de dos formas distintas

Según la tarea a realizar, el espacio de hipótesis será diferente. Por ejemplo, supongamos que queremos saber los días en los que va a llover. Dado un conjunto de datos compuesto por instancias x junto con un valor objetivo $c(x)$, donde $c(x) = 0$ si según las condiciones de x no llueve y $c(x) = 1$ si, según x llueve, el objetivo es estimar c dada una nueva instancia. Se observa que $c : X \rightarrow \{0, 1\}$, donde X es el conjunto de instancias. El conjunto de hipótesis en este caso está formado por todas las hipótesis que el algoritmo puede considerar. De esta forma, $h \in H$ es una función boleana $h : X \rightarrow \{0, 1\}$. Lo que el algoritmo debe aprender es una hipótesis h de tal forma que $h(x) = c(x)\ \forall x \in X$ [11, Pág 23].

Por tanto, el algoritmo tiene que aprender cómo representar los datos bajo las condiciones del espacio de hipótesis. Pero, ¿Cómo se define el aprendizaje?

Según T.Mitchell [11, Pág 2], una definición podría ser la siguiente: Se dice que un programa aprende de una **experiencia**, E, con respecto de una clase de **tareas**, T, y una medida de **rendimiento**, P, si su rendimiento en las tareas de T, medido por P mejora con experiencia E.

A continuación se explicará a qué nos referimos cuando hablamos de tareas, rendimiento y experiencia.

Tareas

Una tarea es un problema que queremos resolver. Lo interesante del *Machine Learning* es que permite resolver problemas que son difíciles de resolver con un código fijo escrito por un humano. Por ejemplo, si queremos enseñar a un robot a hablar, aunque escribamos un programa con todas las reglas del lenguaje, el robot difícilmente pueda conversar como un humano lo hace. Las tareas se diferencian unas de otras según el procesamiento de los ejemplos. Un ejemplo es una serie de características sobre un dato, y estas, normalmente son representadas por las componentes de un vector, $\vec{x} \in \mathbb{R}^n$, [4, Sección 5.1.1]. El vector $\vec{x}$ representa lo que hemos definido como ejemplo y n es el número de características que tiene el ejemplo. Los ejemplos se corresponden con los puntos del conjunto de datos.

Las tareas más comunes que realiza un algoritmo de aprendizaje automático son:

- **Clasificación**: Se le pide al programa dar una función $f : \mathbb{R} \to 1, .., k$ donde k es el número de grupos que hay y f asigna cada ejemplo de entrada a un grupo. Un conjunto de datos muy conocido es el conjunto MNIST que está formado por una gran cantidad de imágenes que representan números del 0 al 9. La tarea de clasificación sobre este conjunto es la de asignar a cada imagen el número que representa.
- **Regresión**:A partir de un conjunto de datos el programa da como salida un valor numérico. La salida esperada del algoritmo es una función $f : \mathbb{R}^n \to \mathbb{R}$.
- **Transcripción**: El sistema observa una representación de ciertos datos y se pide que transcriba la información en formato de texto.
- **Traducción automática**: Cómo su propio nombre indica, dado un texto en cierto idioma, el objetivo es traducir este a un idioma diferente .
- **Detección de anomalías**: En esta tarea el programa se encarga de detectar valores extraños que ocurren en los datos. La aplicación más común de esta tarea es detectar el uso fraudulento de las tarjetas bancarias.
- **Síntesis y muestreo**: Se le pide al algoritmo crear nuevos datos a través de los datos de entrenamiento.
- **Inferencia de valores perdidos**: Puede ser que los valores de entrada sean palabras que le falten algunas letras. La tarea en este caso es predecir que letras son las que faltan.
- **Corrección de errores**: Es común que los datos tengan cierto ruido, es decir, ciertos errores ya bien sean de procesamiento o de medición. El objetivo aquí es quitar esos errores y dar cómo salida un ejemplo sin ruido.

- **Estimación de la densidad de una muestra de datos**: Esta tarea es interesante porque si no conocemos la estructura de los datos, algunas de las tareas anteriores no se pueden realizar. Estimar la función de probabilidad o de densidad (según tengamos variables discretas o continuas) es el paso previo para resolver problemas como predecir valores perdidos. Si $\mathbf{x}_i$ es un valor perdido pero conocemos la distribución de los valores conocidos $\mathbf{x}_j$ entonces podemos calcular la distribución de $\mathbf{x}_i$: $P(\mathbf{x}_i|\mathbf{x}_j)$.

Más ejemplos de tareas e información sobre ellas se pueden encontrar en [4, Pág 97].

Medida de rendimiento

Una medida de rendimiento es imprescindible en un algoritmo de *Machine Learning* porque si no, no podemos medir si nuestro proceso está aprendiendo o no. Es una medida cuantitativa que depende de la tarea que estemos realizando.

Para tareas de clasificación o transcripción se mide la precisión del modelo, es decir, qué proporción de datos ha podido clasificar correctamente. Esta medida corresponde con un número entre 0 y 1. Al igual que se mide la proporción de ejemplos correctamente clasificados, también se puede medir la proporción de ejemplos que dan salidas erróneas. Es lo que se llama como tasa de error. Esta será 0 si el ejemplo está bien clasificado y 1 en caso contrario. Aunque esta medida de rendimiento puede ser útil en algunas tareas, en otras carece de sentido; como por ejemplo en una tarea que genere ejemplos a través de un conjunto inicial de datos [4, Pág 100].

Para poder medir el rendimiento de un algoritmo debemos hacerlo en un conjunto de datos que el algoritmo no haya visto antes. Por tanto si tenemos un conjunto de datos sobre el que queremos hacer cierta tarea debemos dividir el conjunto en dos: Un subconjunto estará formado por los datos de entrada del algoritmo, que se denomina conjunto de entrenamiento o aprendizaje, y el otro subgrupo servirá para probar como funciona nuestro algoritmo y poder medir el error que se produce. Este último se denomina conjunto test.

Elegir una medida de rendimiento es en ocasiones complicado, pues no sabemos qué criterio elegir a la hora de medir el error. Por ejemplo, en la tarea de regresión, ¿es mejor que cometa errores pequeños una gran cantidad de veces o que en ciertos instantes cometa un gran error? La respuesta a esta pregunta dependerá de la aplicación [4, Sección 5.1.2].

Experiencia

Los algoritmos toman experiencia de los datos, y estos se pueden organizar de diferentes formas. Una muy común es la disposición de los datos en una matriz. Como cada punto del conjunto de datos es un vector, podemos poner cada ejemplo en cada fila. De esta forma el conjunto de datos estará formado por tantas filas como datos haya y por tantas columnas como características haya.

Habitualmente los algoritmos de *Machine Learning* se clasifican en aprendizaje supervisado y no supervisado. Los algoritmos, según qué tipo de experiencia tengan permitida durante el proceso de aprendizaje, serán clasificados en un tipo de aprendizaje u otro [4, Pág 101].

En **aprendizaje no supervisado**, los datos que se toman como entrada contienen muchas características. El objetivo es obtener información sobre la estructura de los datos. Una tarea muy común que se enmarca en este campo es el análisis cluster o análisis de conglomerados. El objetivo de un algoritmo de análisis cluster es hacer grupos según las características disponibles del conjunto de datos dado. Otra tarea incluida en este grupo es la estimación de la densidad: a partir de ciertos datos se quiere estimar la distribución de probabilidad que estos siguen [4, Sección 5.1.3].

La diferencia con el **aprendizaje supervisado** es que en este caso el conjunto de datos posee etiquetas de cada ejemplo, es decir, de cada punto del conjunto. Una tarea perteneciente a este tipo de aprendizaje es la tarea de clasificación. Si tenemos un conjunto de datos con etiquetas, tenemos que clasificar el nuevo dato de entrada, $\mathbf{y}$, según las etiquetas que ya tenemos.

Ambos grupos se pueden definir desde un punto de vista matemático. Hemos dicho que los ejemplos son vectores, $\mathbf{x} \in \mathbb{R}^n$. Se pueden interpretar como una muestra proveniente de alguna distribución. En aprendizaje no supervisado se quiere obtener información sobre la estructura de los datos, por tanto, se quiere deducir la distribución de probabilidad, $p(\mathbf{x})$, o algunas características de esta. En el caso del aprendizaje supervisado queremos conocer la distribución condicionada a $\mathbf{x}$ del vector $\mathbf{y}$, $p(\mathbf{y}|\mathbf{x})$, es decir, que estructura tiene el ejemplo $\mathbf{y}$ respecto de los datos $\mathbf{x}$ [4, Pág 103].

Aunque hemos definido dos grupos donde clasificar los algoritmos, estas definiciones permiten cierto grado de flexibilidad. Existen más categorías como el aprendizaje semi-supervisado o el aprendizaje por refuerzo, aunque queda fuera del objetivo principal de este trabajo.

1.2.2 Árboles de decisión

Vemos aquí como ilustración una de las técnicas más conocidas de aprendizaje supervisado, aunque no será el tema principal de estudio en esta memoria.

Una técnica para poder realizar la tarea de clasificación es usar Árboles de Decisión. Esta técnica consiste en clasificar las instancias según sus atributos. Un Árbol de Decisión está compuesto por una raíz de la que salen varios nodos. Cada nodo define una regla de clasificación sobre algún atributo de la instancia. Dependiendo de si nuestra instancia cumple esa regla o no, se tomará un nodo u otro.

Por ejemplo, supongamos que tenemos un conjunto de datos que contiene pacientes con riesgo de tener un ataque cardíaco. Se quiere saber que pacientes tienen un riesgo alto de padecer tal enfermedad y qué pacientes tienen un riesgo bajo. Para ello se construye un árbol con las siguientes preguntas:

- ¿La presión sanguínea sistólica es mayor de 91?
- ¿El paciente es mayor de 62 años?
- ¿El paciente presenta taquicardia sinusal?

Las respuestas a estas preguntas son los atributos del paciente. Dependiendo de si las respuestas son positivas o negativas, el árbol clasificará al paciente x, que sería en este caso una instancia, al grupo de riesgo bajo o al grupo de riesgo alto [5, Pág 519].

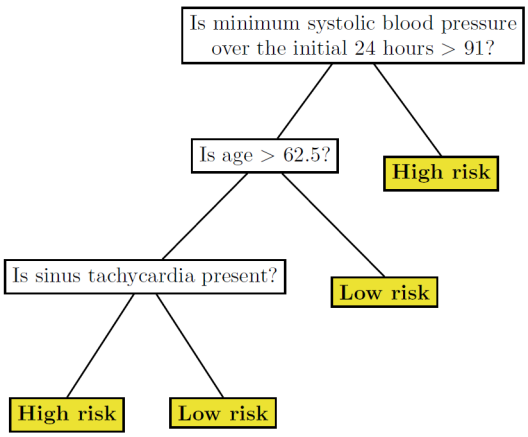


Figura 1.3: Árbol de decisión [5, Pág 519]

El árbol generado en este proceso viene dado en la figura 1.3. Las respuestas positivas a las preguntas se representan en las ramas de la izquierda, y las negativas, en las de la derecha.

En este caso el conjunto de pacientes de riesgo alto viene dado por los pacientes que cumplan las siguientes condiciones:

$$\begin{aligned}&(\text{Presión sistólica de sangre} > 91 \wedge \text{edad} > 62.5 \wedge \text{presenta taquicardia sinusal}) \vee \\ &(\text{Presión sistólica de sangre} \leq 91)\end{aligned}$$

2 Redes neuronales

2.1 Aspectos básicos

Cómo hemos visto el *Deep Learning*, se engloba dentro del *Machine Learning*. En este campo, uno de los objetivos principales es encontrar una buena representación de los datos, tarea difícil en algunos casos. *Deep Learning* solventa este problema haciendo las representaciones paso a paso. En cada paso el programa hace una representación diferente que depende de las anteriores. Las redes neuronales están formadas por capas y estas capas realizan las representaciones antes citadas. La profundidad de la red neuronal viene dada por el número de representaciones que se den en la red.

La red neuronal más básica que existe es el perceptron multicapa (MLP[1]), también llamada red feed forward, de la que hablaremos en la sección 2.2. Esta estructura es una función matemática que transforma los datos de entrada en datos significativos. Esta función se forma por composición de otras, y cada una representa una capa del modelo [4, Pág 5]. Pero, ¿Por qué necesitamos que sea una función matemática? Los programas no entienden los datos extraídos del mundo real. Es por eso por lo que tenemos que representar los datos de entrada mediante una herramienta matemática: Los tensores [3, pág 31] .

2.1.1 Representaciones de datos

Un tensor es una estructura matemática que almacena datos y puede tener diferentes dimensiones. Es una generalización de las matrices y los vectores. Está formado por ejes que contienen datos de cierta dimensión. El número de ejes del tensor da el rango del mismo.

[1] Estas siglas provienen de su traducción al inglés: *Multilayer perceptron.*

- Rango 0:
  Si un tensor tiene rango 0 entonces es un escalar. Es el tensor más básico que existe.
- Rango 1:
  Los vectores son tensores de rango 1. Sin embargo, no hay que confundir un vector de dimensión 5 con un tensor de rango 5. Un vector de dimensión 5 será un tensor de rango 1 que tiene 5 entradas. En cambio, un tensor de rango 5 tiene 5 ejes a lo largo de los cuáles puede tener varias componentes, es decir, cada eje tiene una dimensión diferente. Los tensores de rango 1 representan habitualmente palabras.
- Rango 2:
  Los tensores de rango 2 se pueden identificar con matrices. Por ejemplo una matriz 2x3 se puede interpretar como un tensor de rango 2 donde en su primer eje tiene 2 entradas y en el segundo eje tiene 3 entradas.
  Las imágenes en blanco y negro se pueden representar con tensores de rango 2 donde cada componente representaría la intensidad de gris de un píxel de la imagen. Por tanto, una matriz 3x3 podría representar una imagen en blanco y negro de 9 píxeles, aunque no es la manera habitual de representar imágenes.
- Rango 3 y superior:
  Los tensores de rango 3 se pueden interpretar como un conjunto de matrices. El primer eje nos diría el número filas que tiene cada matriz; el segundo, el número de columnas; y el tercero, el número de matrices que contiene el tensor.
  Si queremos representar una imagen a color podemos guardar información del color de los píxeles en formato RGB (*red-green-blue*). Por tanto necesitamos tres matrices que almacenen la intensidad de rojo, verde y azul de cada píxel respectivamente. Si suponemos que la imagen tiene 256 píxeles podemos representarla con un tensor de rango 3 con dimensión (256,256,3).
  Ahora bien, si queremos representar un vídeo, que no es más que un conjunto de imágenes, necesitamos un tensor de rango 4. En la primera componente tendremos información sobre el número de imágenes que componen el vídeo y en las 3 componentes restantes, la información sobre las imágenes.

Los conjuntos de datos suelen ser muy grandes. El número de ejemplos que contiene un *dataset* suele venir representado en el primer eje del tensor (eje 0). Procesar todos los datos de una vez es un proceso muy costoso, por tanto se suelen coger subconjuntos del dataset. Esto se consigue indicando en el primer eje que puntos extraigo del conjunto de datos. Por ejemplo, si tenemos un conjunto de imágenes representadas en el tensor de rango 4 con dimensiones (1000,256,256,3) quiere decir que tenemos 1000

imágenes a color de 256x256 píxeles. Un subconjunto del dataset de 100 imágenes, más comúnmente llamado ***batch***, viene representado por (100,256,256,3). Para trabajar con todos los ejemplos, dividimos el conjunto de datos en varios *batches*, y de esta forma la red neuronal procesa los datos poco a poco [3, Sección 2.2].

En resumen, según la tarea a realizar necesitaremos representar los datos de una manera u otra. Dependiendo del tipo de dato y tarea, usaremos tensores de diferentes rangos y dimensiones.

2.1.2 Operaciones tensoriales

Las redes neuronales aprenden a representar los datos a través de las operaciones tensoriales.

Operaciones por componentes

La suma y la operación *ReLu* (*Rectified Lineal Unit*) son operaciones que se hacen componente a componente. La operación *ReLu* se define como: $ReLu(x) = \text{máx}(0, x)$. Esta operación es una función de activación habitual. Las funciones de activación se estudiarán más adelante.

La operación suma se realiza entre tensores que tienen el mismo rango. La operación consiste en sumar componente a componente cada tensor.

Broadcasting

Si queremos sumar dos tensores que no tienen el mismo rango o la misma dimensión en sus ejes usamos esta operación. El tensor de rango más pequeño se convertirá en otro que tenga las mismas dimensiones que el tensor de rango mayor. *Broadcasting* (transmitir, en castellano) se realiza en dos pasos. Primero añadimos los ejes necesarios al tensor más pequeño hasta que tenga el mismo número de ejes que el otro tensor. Finalmente, el tensor se repite a lo largo de los nuevos ejes. Es necesario tener esta operación en cuenta, pero en la práctica, según el programa que utilicemos, esta operación se realizará de forma automática si es necesario [3, Sección 2.3.2].

Producto escalar

El producto escalar de dos tensores coincide con el producto escalar para vectores y con el producto matricial para matrices. Las dimensiones del último eje del primer tensor y del primer eje del segundo tensor deben coincidir. Es una generalización del producto matricial. Dos matrices no se pueden multiplicar si no coincide el número de columnas de la primera con el número de filas de la segunda. El resultado de multiplicar una matriz 2x3 por una matriz 3x4 será una matriz 2x4. En tensores si tenemos un tensor de rango 5 de dimensiones (a,b,c,d,e) y otro de rango 3 con dimensiones (f,g,h) entonces f debe coincidir con e para poder realizar el producto escalar; y el resultado es un tensor de rango 6 de dimensiones (a,b,c,d,g,h).

Reshape

Esta operación sirve para cambiar la forma del tensor. Por ejemplo, un vector de dimensión 9 podemos verlo como una matriz 3x3.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Análogamente, podemos ajustar las dimensiones y los ejes del tensor para que tenga una forma concreta. Esta operación es útil cuando queremos que los tensores tengan una forma específica que encaje con nuestro modelo [3, Sección 2.3]

2.2 Red neuronal feed-forward

La red neuronal feedforward, o lo que es lo mismo, el perceptrón multicapa, es uno de los tipos más básicos y, a la vez, más útiles en el ámbito del *Deep Learning*. El objetivo de este tipo de red neuronal es aproximar una función f^*. Esta red define una aplicación $y = f(x, \theta)$ donde x son los datos de entrada, y es la salida generada por la red y θ es un conjunto de parámetros que la red neuronal va aprendiendo a modificar según el objetivo final.

Estas redes neuronales están formadas por distintos tipos de capas: La capa de entrada, la capa de salida y las capas ocultas. Cada capa, i, viene representada por una función, f_i. Estas funciones se componen de manera adecuada y dan como resultado la

función $f = f_k(...(f_3(f_2(f_1))))$ que será la que transforma nuestros datos de entrada, x, en la salida, y [4, Pág 163].

La red se llama "neuronal" por su parecido a las neuronas de nuestro cerebro. Las funciones que forman las capas son funciones tensoriales. Entonces podemos interpretar que cada componente de la función es una neurona. Con está interpretación tenemos que las capas están formadas por neuronas y cada capa se conecta con las neuronas de la siguiente capa, tal y como se hace en la composición de funciones. Las funciones de las capas deben cumplir una condición para que la red funcione, y es que si $f_i : \mathbb{R}^n \to \mathbb{R}^m$ entonces la función $f_{i+1} : \mathbb{R}^m \to \mathbb{R}^l$ para que tengan sentido la operación de composición de funciones. Dicho de otro modo, el número de componentes del vector de salida de la capa i debe coincidir con el número de componentes del vector de entrada de la capa $i+1$. En [3, Pág 47] se da una definición a *grosso modo* de lo que son las redes neuronales:

> Las redes neuronales no son más que una cadena de operaciones tensoriales, y estas operaciones tensoriales no son más que transformaciones geométricas de los datos de entrada.

A continuación, se explica cómo están formadas estas funciones f_i. Supongamos que tenemos como entrada un tensor, x. Recordemos que el objetivo es representar este dato de entrada de una forma diferente. Para eso usamos transformaciones afines que son operaciones de la forma $Wx+b$, donde W y b son también tensores. Estos son los atributos de la capa. W se denomina *kernel* y b se denomina *bias*. Supongamos que nuestra red tiene 2 capas. La primera capa viene representada por la transformación afín $f_1(x) = W_1x + b_1$ y la segunda capa por $f_2(x) = W_2x + b_2$. Cómo hemos dicho la red es la composición de estas funciones. Entonces la salida de la red sería $y = f_2(W_1x + b_1) = W_2(W_1x + b_1) + b_2$, que sigue siendo una transformación afín. Por tanto, no tiene sentido que las capas sean transformaciones afines de los datos, pues una red de 10 capas sería lo mismo que una red con una única capa. El problema es que falta un ingrediente esencial para que las redes consigan representaciones útiles de los datos: Las **funciones de activación** [3, Pág 46].

Las funciones de activación son funciones no lineales que sirven para que las capas obtengan representaciones más ricas de los datos, es decir, que obtengan representaciones más allá de las transformaciones afines[3, Pág 101]. Las funciones de activación más comunes son [18]:

- *ReLu*: El nombre de esta función proviene de su traducción al inglés, *Rectified Linear Unit*. Su función viene dada por

$$ReLu(x) = \text{máx}(0, x)$$

- Sigmoide: Esta función es comúnmente representada por la letra σ. La imagen de esta función está contenida en el intervalo [0,1] y su expresión viene dada por [4, Sección 3.10]:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Tanh: Es la función tangente hiperbólica. Toma valores en el intervalo [-1,1]. Es parecida a la función sigmoide.

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Softmax: Esta función se conoce como la exponencial generalizada. Sea $x = (x_1, ..., x_k)$. Entonces la componente i-ésima de la función se define como:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

  A diferencia de las otras funciones de activación, esta no se aplica componente a componente, sino que se aplica a un vector; en nuestro caso a la salida de una capa.

En un principio los parámetros de la capa, W y b, vendrán dados de forma aleatoria, lo que nos lleva a representaciones de datos que carecen de sentido. Para obtener representaciones útiles de los datos tenemos que ir cambiando los pesos. Estos cambios se hacen a través de un entrenamiento. Para ello necesitamos un optimizador, una función de pérdida y una medida de precisión. Además debemos recordar que cada valor del conjunto de datos de entrada tiene asociado otro valor, y_{target}, qué digamos qué es el valor objetivo.

Si insertamos unos datos de entrada en la red, nos dará unos datos de salida, que se pueden entender como una predicción de nuestros datos, y_{pred}. Con la función de pérdida podemos ver cómo de alejados están las predicciones de los valores objetivo y_{target}. El objetivo es minimizar el valor de la función de pérdida, entonces debemos modificar un poco los pesos de la red, W y b de forma que se minimice la función de pérdida. Este proceso se denomina "bucle de entrenamiento" (*training loop*). El entrenamiento terminará cuando el valor de la función de coste sea muy pequeño. Para

minimizar la función de pérdida debemos resolver un problema de optimización y para ello hace falta herramientas matemáticas.

Las funciones de las capas son continuas por lo que un pequeño cambio en los atributos de la capa provoca un pequeño cambio en la salida final. Además, la funciones son diferenciables porque son composición de funciones más básicas, todas ellas diferenciables (excepto la función de activación *ReLu* que en el cero no es diferenciable). Por tanto, cómo cambia la función respecto de los parámetros se puede calcular a través del gradiente de la función. En conclusión, usaremos el algoritmo del gradiente para actualizar los parámetros de las capas de manera que se minimice la función de pérdida [3, Sección 2.4].

2.2.1 Algoritmo del descenso de gradiente

Supongamos que nuestra función de pérdida es E. Esta función mide cuánto se diferencian y_{pred} de y_{target}. Este último tensor es fijo. Sin embargo, y_{pred} dependerá de los parámetros de la red. Si denominamos a esos parámetros W entonces podemos escribir $E(W)$. Usaremos el gradiente para modificar W. Supongamos que el valor actual de W es W_0. Sea $\nabla E(W_0)$ el gradiente en W_0. Esta función representa la dirección máxima de ascenso alrededor de W_0. Entonces debemos mover W en dirección contraria a la del gradiente. Por tanto los nuevos valores son $W_1 = W_0 + \alpha_0 \nabla E(W_0)$ donde α_0 es una cantidad pequeña que dice cuánto nos debemos de mover. Hacer este proceso reiteradamente es lo que se conoce como el algoritmo del descenso de gradiente. Si nos encontramos en la etapa k, entonces $W_{k+1} = W_k + \alpha_k \nabla E(W_k)$. El algoritmo acaba cuando $\nabla E(W_k) = 0$ [3, Pág 52].

El parámetro α se denomina factor de aprendizaje y debemos elegirlo cuidadosamente. Un factor de aprendizaje muy pequeño producirá muchas iteraciones y nos podríamos estancar en un mínimo local. En cambio, un factor de aprendizaje grande implica un gran desplazamiento y puede descontrolar los valores que se toman en cada etapa.

Debido a que la función de pérdida depende de y_{pred} y esta tiene una expresión que es composición de funciones de activación y operaciones tensoriales sobre los parámetros respecto de los que derivamos, es a veces complicado calcular el gradiente de la función de pérdida. Para calcularlo de manera eficiente usamos el Algoritmo de retropropagación (*backpropagation*).

2.2.2 Algoritmo *backpropagation*

Este algoritmo se basa en la regla de la cadena para calcular el gradiente de una función.

Teorema 2.1 (Regla de la cadena). *Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función diferenciable en el punto $x_0 \in \mathbb{R}^n$, y sea $g : \mathbb{R}^m \to \mathbb{R}^n$ una función diferenciable en el punto $u_0 \in \mathbb{R}^m$ con $x_0 = g(u_0)$. Entonces la composición, $h = f \circ g$ es diferenciable en u_0 y*

$$Dh(u_0) = D(f \circ g)(u_0) = Df(g(u_0))Dg(u_0)$$

donde $Df(g(u_0))$ es el gradiente de f en x_0 y $Dg(u_0)$ es la matriz jacobiana de g en u_0. [20, Pág 340]

Hemos visto este teorema en su forma vectorial. Sin embargo, en las redes neuronales es más común trabajar con tensores. Esto no supone ningún problema porque podemos distribuir las entradas del tensor en forma de vector y calcular su gradiente de manera idéntica a cómo se hace con vectores. Una vez calculado hacemos la operación de "reshape" para devolver el tensor a su forma original [3, Sección 2.4.2].

Supongamos que tenemos una red neuronal feed-forward donde n_{in} es el número de unidades de entrada, n_h, número de unidades de las capas intermedias y n_{out} el de las unidades de salida. También se necesita una función de coste o de pérdida. La más común es el error medio cuadrático:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$$

donde D es el conjunto de los datos de entrenamiento, *outputs* es el conjunto de unidades de salida de la red, y t_{kd} y o_{kd} son los valores target y output de la k-ésima unidad de salida donde se está usando el dato de entrenamiento d.

Para explicar el algoritmo de backpropagation es necesario dar notación a los elementos que este usa. Denotaremos x_{ji} a la entrada i-ésima de la unidad j, w_{ji} al peso asociado a la conexión de la neurona i a la neurona j, net_j es la suma ponderada de las entradas a la unidad j,es decir, $\text{net}_j = \sum_i w_{ji} x_{ji}$, o_j es el valor de salida de la neurona j y t_j es el valor objetivo de la neurona j. Si calculamos el gradiente de la función de coste para cada dato $d \in D$ podremos realizar el algoritmo del descenso del gradiente. Denotamos $E_d(\vec{w}) = \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$ a la función de coste asociada al dato de

entrada d. El objetivo es calcular $\frac{\partial E_d}{\partial w_{ji}}$. w_{ji} cambia el valor de la red a través del valor de net_j. Aplicando la regla de la cadena obtenemos:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}}$$

Ahora bien $\text{net}_j = \sum_i w_{ji} x_{ji}$. Es claro que $\frac{\partial \text{net}_j}{\partial w_{ji}} = x_{ji}$. Por tanto queda calcular $\frac{\partial E_d}{\partial \text{net}_j}$, que tendrá una expresión diferente dependiendo si las unidades de las que estamos calculando su error es de salida o no.

Error en las unidades de salida

El error en estas neuronas dependerá del valor de salida o_j. Por tanto, aplicando de nuevo la regla de la cadena:

$$\frac{\partial E_d}{\partial \text{net}_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j}$$

Es fácil comprobar que

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2 = -(t_j - o_j)$$

Recordamos que $o_j = a(\text{net}_j)$ donde a denota una función de activación cualquiera. Como consecuencia,

$$\frac{\partial o_j}{\partial \text{net}_j} = \frac{\partial a(\text{net}_j)}{\partial \text{net}_j} = a'(\text{net}_j)$$

El cálculo de la expresión $\frac{\partial E_d}{\partial w_{ji}}$ viene dado por:

$$\frac{\partial E_d}{\partial w_{ji}} = -(t_j - o_j) a'(\text{net}_j) x_j i$$

Error en las unidades ocultas

Para cambiar los pesos de las capas ocultas debemos tener en cuenta todas las conexiones que tiene una unidad oculta dentro de la red. Es por ello por lo que se define un conjunto que contenga a todas las unidades que estén conectadas con la unidad j

en la capa anterior a la que se encuentra j. Este conjunto se denotará $downstream(j)$. Además, también se define el error de cada unidad de salida como $\delta_k = -\frac{\partial E_d}{\partial \text{net}_k}$. Por consiguiente:

$$\begin{aligned}\frac{\partial E_d}{\partial \text{net}_j} = \\ &= \sum_{k \in downstream(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} = \\ &= \sum_{k \in downstream(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} = \\ &= \sum_{k \in downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial \text{net}_j} = \\ &= \sum_{k \in downstream(j)} -\delta_k w_{kj} a'(\text{net}_j)\end{aligned}$$

Como $net_k = \sum_i w_{ki} x_{ki}$, la derivada de esta expresión respecto de o_j será 0 en todos los sumandos excepto cuando $i = j$. En este caso x_{kj} coincide con o_j; pues la salida de la unidad j es igual que la entrada de la unidad k cuando esta proviene de la unidad j. En cuánto al cálculo de $\frac{\partial o_j}{\partial \text{net}_j}$, es equivalente al caso de las unidades de la capa de salida. Por tanto,

$$\frac{\partial E_d}{\partial w_{ji}} = -a'(\text{net}_j) \sum_{k \in downstream(j)} \delta_k w_{kj} x_{ji}$$

Con esto ya tenemos los ingredientes necesarios para el desarrollo del algoritmo del descenso del gradiente usando el algoritmo de backpropagation.

Algoritmo de Backpropagation

- Supongamos que la red neuronal feed-forward tiene n_{in} unidades de entrada, n_h unidades ocultas y n_{out} unidades de salida.
- Calcular la salida de la red con pesos iniciales aleatorios.
- Hasta que se cumpla la condición de finalización[2], hacer para cada dato de entrenamiento (x, y):
  Propagar la entrada x a través de la red
  1. Dado una instancia x calcular el valor de salida de cada unidad de la red.

[2]En este caso, esta condición equivale a que la función de coste alcance un valor pequeño dado.

Propagar los errores hacia atrás en la red

2. Para cada unidad de salida k calcular:

$$\delta_k = -\frac{\partial E_d}{\partial \text{net}_k} = (t_k - o_k)a'(net_k)$$

3. Para cada unidad oculta h calcular

$$\delta_h = -\frac{\partial E_d}{\partial \text{net}_h} = -a'(\text{net}_h) \sum_{k \in downstream(h)} \delta_k w_{kh}$$

4. Actualizar los pesos de la red usando el algoritmo del descenso del gradiente:

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

con

$$\Delta w_{ji} = \alpha \delta_j x_{ji}$$

donde α es el factor de aprendizaje en el algoritmo del gradiente.

La referencia de esta sección es [11, Págs. 98-103]

2.3 Redes recurrentes

Un problema que tienen las redes neuronales *feed-forward* es que no tienen capacidad para recordar características de los datos cuando pasamos de una capa a otra. Las redes neuronales recurrentes (*Recurrent Neural Networks*, **RNN**) intentan solventar este problema. Las RNN son redes *feed-forward* conectadas entre sí donde la salida de una capa se toma como entrada en la siguiente. Se puede interpretar como muchas copias de una misma red donde la información va fluyendo de una red a su sucesora, como podemos ver en la figura 2.1.

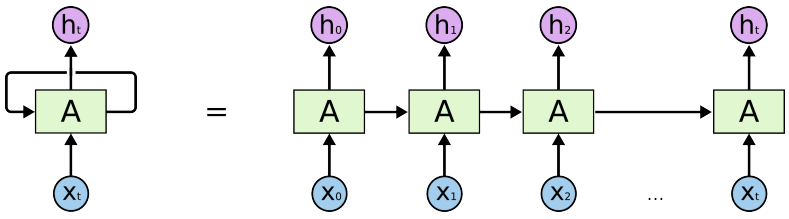


Figura 2.1: Red Neuronal Recurrente. Blog de C.Olah[15]

Cada x_i se refiere a los datos de entrada en el paso i; y cada h_i es la salida de la red neuronal "A" en el paso i. En cada paso, la red neuronal "A" toma como entrada dos valores: los datos de entrada correspondientes en ese paso, x_i, y la salida de la red en el paso anterior, h_{i-1}.

Este tipo de redes se suelen usar con datos que tienen una estructura de cadena o lista, como por ejemplo secuencias de texto. Podemos usar esta estructura para predecir espacios en una oración. Por ejemplo en la oración "*La Tierra gira alrededor del .....*" la red debería dar como salida la palabra "*Sol*". Quizás en este caso la red pueda actuar sin grandes problemas. Sin embargo, si queremos predecir una palabra que se encuentra en un texto de varias oraciones es posible que el contexto que necesitamos para la predicción se encuentre varias líneas atrás. Este es un problema conocido como dependencia a largo plazo y las RNN no son capaces de resolverlo. Es esto lo que motiva a la construcción de un nuevo tipo de RNN : las redes **LSTM** [15].

2.3.1 Redes LSTM

Las redes **Long Short Term Memory**, más comúnmente llamadas LSTM, son un tipo de red recurrente que nace con el objetivo de resolver el problema de la dependencia a largo plazo. La estructura de este tipo de red es igual que el de la figura 2.1, pero el bloque "A" está formado por varias componentes que tienen diferentes funciones. Veamos cómo se construyen.

La componente principal de la red es la célula de memoria, la cual contiene la información del módulo anterior. Podemos quitar o añadir información de la célula. La cantidad que se añade o se quita viene dada por lo que se denominan puertas, y estas están compuestas por capas sigmoides. En total hay 3 puertas: una que olvida información, otra que añade y una última qué decide la salida final. Todas toman como entrada a x_t, entrada en ese paso, y a h_{t-1}, salida del módulo anterior, y además todas tienen una estructura parecida [23, Sección 10.1].

- **Puerta del olvido**:

$$F_t(x_t, h_{t-1}) = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

- **Puerta de entrada**:

$$I_t(x_t, h_{t-1}) = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

- **Puerta de salida**:

$$O_t(x_t, h_{t-1}) = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

W_{xj},W_{hj} y b_j con $j \in \{f, i, o\}$ representan los atributos kernel y bias de cada capa.

Con estas capas podemos obtener el valor de salida de la célula de memoria. En primer lugar, multiplicamos la información de la célula de memoria interna procedente del paso $t-1$ por I_t y de esta forma olvidamos la información dictada por la puerta del olvido. Para ver que información nueva añadimos necesitamos dos componentes, la puerta de entrada, I_t y una capa con función de activación tanh que denotaremos $\tilde{C}_t$. Esta última crea un vector que dice qué cantidad de información de I_t va a pasar a la célula de memoria. $\tilde{C}_t$ es de la forma:

$$\tilde{C}_t(x_t, h_{t-1}) = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

Haciendo el producto escalar de I_t por $\tilde{C}_t$ obtenemos la cantidad de información que queremos añadir a la célula de memoria interna. Por tanto la nueva célula de memoria viene dada por:

$$C_t = F_t \cdot C_{t-1} + I_t \cdot \tilde{C}_t$$

Por último, debemos decidir que información forma la salida de este módulo, h_t. Es aquí dónde entra en juego la puerta de salida. Primero aplicamos una capa con función de activación tanh a la célula de estado, C_t. Esto nos da valores entre -1 y 1. A continuación hacemos el producto escalar de la puerta de salida, O_t, que nos dice con qué cantidad de información de los valores de entrada nos quedamos, con el resultado de $\tanh(C_t)$. Por tanto la salida será de la forma:

$$h_t = O_t(x_t, h_{t-1}) \cdot \tanh(C_t)$$

Esta estructura se va repitiendo paso a paso a lo largo de la red recurrente. Según qué información se olvida y qué información se añade, la red LSTM va creando distintas salidas. En la figura 2.2 se muestra la estructura interna de cada módulo [15].

Cuando tenemos como entrada un texto, la red LSTM va leyendo el texto palabra por palabra de manera secuencial, de tal forma que en cada paso la red lee una palabra nueva de la secuencia. Se observa que en cada paso, los estados intermedios de la red neuronal, h_i, se van modificando con la información proveniente de la etapa anterior,

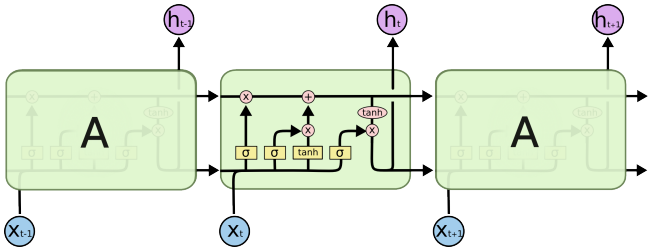


Figura 2.2: Estructura de la LSTM.Blog de C.Olah[15]

h_{i-1}, y la nueva palabra, x_i. La salida final de la red es h_T donde T es el número total de etapas, es decir, la salida es el estado final de la red LSTM. Por tanto, los valores intermedios h_i se olvidan, provocando así una pérdida de información que podría ser útil en ciertas tareas. Con la llegada del mecanismo de atención, veremos que este modelo puede ser mejorado guardando la cantidad necesaria de información de cada paso intermedio h_i.

3 El mecanismo de atención

3.1 Introducción

El mecanismo de atención nace como una mejora en las tareas de traducción realizadas por las redes recurrentes y las redes LSTM. El problema de estos modelos venía cuando las frases a traducir eran demasiado largas y la red no era capaz de que la información persistiera en el tiempo. Una nueva idea fue dada en [2]. En el instante de traducir una nueva palabra tendremos en cuenta la información de las palabras precedentes a esta. Cada término de la oración de entrada tendrá cierta importancia a la hora de traducir. Quién determina esa importancia es el **mecanismo de atención**. Usando como ejemplo la tarea de traducción se desarrollará el funcionamiento del mecanismo de atención, que después puede ser usado en otros campos e implementado en diferentes estructuras de redes neuronales.

3.2 Componentes principales

Supongamos que vamos a realizar una tarea de traducción inglés-español. Una idea sería buscar la palabra que queremos traducir en un diccionario que contenga palabras en inglés junto con su traducción al español. Llamaremos claves a las palabras del diccionario que están en inglés y valores a su traducción en español. La palabra que queremos traducir se denotará como consulta. Las palabras vendrán representadas por vectores. Habitualmente estos vectores se denotarán *query*, $\mathbf{q}$, a aquello que vamos a consultar, *keys*, $\mathbf{k}$, a las palabras que son las claves del diccionario y *values*, $\mathbf{v}$, a los valores correspondientes a tales *keys*. Con este ejemplo podemos entender los conceptos de *queries*, *keys* y *values*, pero más generalmente estos vectores no solo estarán representando palabras, si no que pueden representar cualquier tipo de dato

con el que vayamos a trabajar.

Teniendo esto en cuenta podemos dar la siguiente definición [23, Sección 11.1]:

Definición 3.1 (Diccionario). *Un diccionario es un conjunto de pares donde el primer elemento es una clave del diccionario, $\boldsymbol{k}$, y el segundo elemento es el valor, $\boldsymbol{v}$, asociado a esa clave.*

La forma tradicional de trabajar con un diccionario es bien conocida. Dado una palabra (*query*) se busca en el diccionario, comparando cada clave del diccionario con la consulta a realizar, y devolvemos el valor asociado a la clave que sea igual que nuestra consulta. Pero ¿que pasaría si nuestro *query* no está en el diccionario? Quizás no exista ninguna clave del diccionario que sea idéntica a nuestra consulta pero puede ser que el vector *query* guarde cierta relación con varias claves del diccionario. El resultado de hacer la consulta $\mathbf{q}$ en el diccionario $\mathcal{D}$ será una suma ponderada de los vectores *values* donde la ponderación la da una función escalar, $\alpha(\cdot,\cdot)$. Esta función tiene como entrada dos vectores ($\mathbf{q}$ y $\mathbf{k}$) y da como resultado un número que indica la relación entre ambos vectores. Para cada $\mathbf{k}$ tendremos una puntuación diferente. Estos valores se denotarán **pesos de atención**. La combinación lineal (o suma ponderada) descrita es lo que se conoce como **Mecanismo de atención**. A continuación se mostrará el mecanismo de atención de forma matemática [23, Sección 11.1]:

Definición 3.2 (Mecanismo de atención). *Se define la atención como:*

$$Attention(\boldsymbol{q}, \mathcal{D}) = \sum_{i=1}^{m} \alpha(\boldsymbol{q}, \boldsymbol{k}_i)\boldsymbol{v}_i \tag{3.1}$$

Donde $\boldsymbol{q} \in \mathbb{R}^k$ es el vector query, $\mathcal{D}$ un diccionario, $\boldsymbol{k}_i \in \mathbb{R}^k$ y $\boldsymbol{v}_i \in \mathbb{R}^v$ las claves y los valores del diccionario respectivamente, m el número de pares del diccionario $\mathcal{D}$ y $\alpha(\cdot,\cdot)$ una función que representa la relación entre dos vectores dados. Esta función denota los pesos de atención. Se observa que los vectores query y key tienen que tener la misma dimensión, k. Sin embargo, los vectores que representan los valores del diccionario pueden tener distinta dimensión, v.

Para valores altos de $\alpha(\cdot,\cdot)$, los vectores $\mathbf{q}$ y $\mathbf{k}$ estarán muy relacionados. Es por eso por el que recibe el nombre de atención, pues los $\mathbf{v}$ asociados a los vectores $\mathbf{k}$ que tienen pesos altos de α se le presta más importancia o atención que a los $\mathbf{k}$ que tienen valores bajos en esta función. Si el vector $\mathbf{q}$ que estamos consultando coincide con algunos de los vectores $\mathbf{k}$ que pertenecen al diccionario, entonces el peso asociado a

tal $\mathbf{k}$ deberá ser 1 y el peso asociado a los demás $\mathbf{k}$ deberá ser 0. Este sería el caso en el que nuestra consulta está en el diccionario. El resultado de atención sería el valor, $\mathbf{v}$ asociado al $\mathbf{k}$ que tiene como peso 1, pues al hacer la suma ponderada solo habría un valor que sumar: El correspondiente a tal $\mathbf{k}$.

El mecanismo de atención habitualmente es una combinación convexa de los valores, $\mathbf{v}$. Es decir, los pesos de atención tienen que cumplir que $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0 \; \forall i$ y $\sum_i^m \alpha(\mathbf{q}, \mathbf{k}_i) = 1$,[23, Sección 11.1]

Por tanto, para calcular los pesos de atención necesitamos [13]:

- $e = f(\mathbf{q}, \mathbf{k})$: Función que calcula la relación entre los vectores $\mathbf{q}$ y $\mathbf{k}$. Está función se denota como función de puntuación,aunque también es conocida como función de energía ,de compatibilidad o de similitud.
- $g(e)$: Función de distribución. Dados las puntuaciones, $\mathbf{e}$, las normaliza de tal forma que las puntuaciones sigan una función de distribución, es decir, la suma de todas será 1 y cada una de ellas será mayor o igual a 0.

Con estas dos funciones podemos calcular los distintos pesos de atención:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = g(f(\mathbf{q}, \mathbf{k}_i)) \tag{3.2}$$

La función más usada para transformar los resultados de la función de puntuación en una distribución es la función *softmax*. Aunque es la más común, hay diversos trabajos donde se proponen nuevas funciones de distribución que podrían mejorar a esta dependiendo de las tareas a realizar, como por ejemplo en [9]. Sin embargo, no entraremos en detalle, ya que en nuestro caso usaremos siempre la función *softmax*.

A continuación se estudiará diferentes funciones de puntuación.

3.3 Funciones de puntuación

3.3.1 Funciones kernel

Una forma de medir distancias es a través de funciones kernel. Las distancias se miden a través de estas funciones gracias a un proceso que se denomina *kernel trick*.

Consideremos un conjunto de datos de entrenamiento $(x_1, y_1), \ldots, (x_m, y_m) \in \mathcal{X} \times \mathcal{Y}$ donde $\mathcal{X}$ es el conjunto de instancias, y $\mathcal{Y}$ el conjunto de las etiquetas. Nuestro objetivo es calcular un elemento $y \in \mathcal{Y}$ cuando recibimos como entrada un elemento $x \in \mathcal{X}$ que esté fuera del conjunto de entrenamiento. La idea es que si podemos medir la distancia de x a los distintos $x_i \in \mathcal{X}$, la salida y será parecida a y_j donde x_j es el elemento que más se parece a x. El problema es que medir la similitud de dos objetos de $\mathcal{X}$ puede ser subjetiva. La solución es transformar los elementos de $\mathcal{X}$ en un espacio de características de $\mathcal{X}$, denominado F, a través de una función $\phi : \mathcal{X} \to F$. F es un espacio euclídeo pero de gran dimensión. Por tanto podemos calcular su producto escalar asociado. De esta forma podemos calcular la similitud de $x, x' \in \mathcal{X}$ haciendo $\langle \phi(x), \phi(x') \rangle$. Dado que F tiene gran dimensión, se busca un subespacio donde el producto pueda ser calculado a través de una función no lineal denominada kernel:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{3.3}$$

Esto es lo que se conoce cómo *kernel trick* [17].

La cuestión es ahora saber que funciones kernel cumplen la propiedad anterior. Unos que funcionan bien son los kernel definidos positivos.

Definición 3.3 (Kernel definido positivo). *Se denomina kernel definido positivo (kdf) a una función simétrica $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ tal que $\forall m \in \mathbb{N}, x_i \in \mathcal{X}$, da lugar a una matriz de Gram definida positiva, es decir, $\forall c_i \in \mathbb{R}$ se tiene*

$$\sum_{i,j=1}^{m} c_i c_j k_{ij} \geq 0 \; donde \; k_{ij} = k(x_i, x_j)$$

A través de estas funciones deberíamos ser capaces de construir un espacio en el cual el producto escalar de ese espacio nos dé el producto escalar de $\langle \phi(x), \phi(x') \rangle$ antes definido. En este caso construiremos un espacio de funciones dotado de un producto escalar de la siguiente forma:

1. Definimos la aplicación $\phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}$ que lleva $x \mapsto k(\cdot, x)$ donde $\mathbb{R}^{\mathcal{X}}$ es el espacio de funciones que transforman $\mathcal{X}$ en $\mathbb{R}$, lo que es lo mismo, el dual de $\mathcal{X}$.
2. A este espacio le añadimos las combinaciones lineales de estas funciones para convertirlo en un espacio vectorial:

$$f(\cdot) = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i)$$

$$g(\cdot) = \sum_{i=1}^{m'} \beta_j k(\cdot, x'_j)$$

donde $m, m' \in \mathbb{N}; \alpha_i, \beta_j \in \mathbb{R}; x_i, x_j \in \mathcal{X}$. f y g son funciones formadas por combinaciones lineales de nuestro espacio de partida.

3. Definir un producto escalar en este espacio:

$$\langle f, g \rangle := \sum_{i=1}^{m} \sum_{j=1}^{m'} \alpha_i \beta_j k(x_i, x'_j)$$

Con esto ya tendríamos un espacio pre-hilbertiano. Faltaría convertirlo a un espacio Hilbert, H_k, completando la norma.

A través de esta construcción tenemos la siguiente igualdad:

$$\langle \phi(x), \phi(x') \rangle = \langle k(\cdot, x), k(\cdot, x') \rangle = k(x, x')$$

tal y como queríamos [17].

En el ámbito del mecanismo de atención, en [23, Sección 11.2] se propone utilizar distintos tipos de kernel que satisfacen las ideas expuestas. En nuestro contexto sustituimos los valores $x \in \mathcal{X}$ por los queries y values, que son los elementos que queremos relacionar, o dicho de otro modo, calcular una similaridad. A continuación tenemos dos ejemplos de funciones kernel.

- Gaussiano:
$$K(\mathbf{q}, \mathbf{k}) = exp(-\frac{1}{2}||\mathbf{q} - \mathbf{k}||^2)$$
- Boxcar:
$$K(\mathbf{q}, \mathbf{k}) = \frac{1}{2}\mathbf{I}(||\mathbf{q} - \mathbf{k}|| \leq 1)$$

donde $\mathbf{I}(x \leq 1) = \begin{cases} 1 & \text{si } x \leq 1 \\ 0 & \text{si } x > 1 \end{cases}$.

Las funciones kernel son útiles para algunas tareas. Sin embargo, si queremos medir distancias entre imágenes quizás sea mejor usar otro tipo de funciones de puntuación.

3.3.2 Métricas

Dada una base de datos de imágenes, una tarea podría ser dar como salida una imagen que tenga características similares a una imagen dada. Para esta tarea en [12] se proponen distancias métricas, que son habituales en un curso de topología básica. A través de estas métricas mediremos la distancia entre los distintos puntos de la imagen de entrada con las imágenes del conjunto de datos.

- Distancia euclídea:
  La distancia euclídea se define como $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ donde x, y son vectores de $\mathbb{R}^n$. Es la distancia habitual en $\mathbb{R}^n$.
- Distancia euclídea estándar:
  Suponiendo que nuestros datos siguen una distribución, podemos calcular su media y su desviación típica, σ. Si tipificamos los datos definimos si $= \frac{\text{dato original-media}}{\sigma}$.
  Se define la distancia euclídea tipificada como:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}\left(\frac{x_i - y_i}{si}\right)^2}$$

- Distancia de Minkowski:

$$d(x, y) = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{\frac{1}{p}}$$

  Para $p = 2$ obtenemos la distancia euclídea y para $p = 1$, la distancia Manhattan, que se define a continuación.
- Distancia Manhattan o distancia taxi:
  La distancia Manhattan se define como $d_m(x, y) = \sum_{i=1}^{n}|x_i - y_i|$. Esta distancia se obtiene cuando en la distancia de Minkowski $p = 1$. Es por eso por lo que también se denomina distancia L^1. Si suponemos que nuestros puntos están en una cuadrícula, la distancia Manhattan será la suma de los segmentos contenidos en la cuadrícula que necesitamos para ir de un punto a otro. Podemos ver la diferencia entre la distancia euclídea y la distancia taxi en 3.1

- Distancia de Chebyshev:
  La distancia de Chebyshev se escribe como $d(x, y) = \text{máx}_{i=1\ldots n}(|x_i - y_i|)$ Esta distancia se obtiene mediante la distancia de Minkowski cuando $p \to \infty$. Un

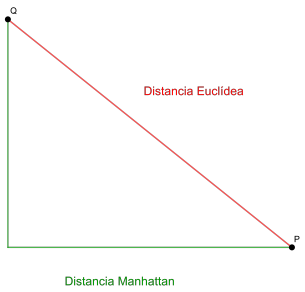


Figura 3.1: Distancia euclídea y distancia Manhattan

término muy habitual en topología son las bolas:

$$B(x,r) = \{p \in \mathbb{R}^n | d(x,p) < r\}$$

Una bola no es más que el conjunto de puntos que están a distancia r de un punto dado, x. Una propiedad importante de la distancia Chebyshev es que las bolas en esta distancia son cuadradas.

- Distancia de Mahalanobis:
  Supongamos que nuestro conjunto de datos sigue una distribución con media μ y matriz de covarianzas Σ. La distancia de Mahalanobis mide la distancia entre un punto (la imagen de entrada) y una distribución (la distribución de los datos). Se define como: $d_M(x) = \sqrt{(x-\mu)^t \Sigma^{-1}(x-\mu)}$
  También se puede entender como una medida de disimilaridad.

Definición 3.4 (disimilaridad). *Sea $\mathcal{O}$ un conjunto de datos. Una disimilaridad es una aplicación $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}$ que cumple:*

- $d(O_i, O_j) \geq 0$
- $d(O_i, O_j) = 0 \iff i = j$
- $d(O_i, O_j) = d(O_j, O_i)$

Dados dos vectores aleatorios $\vec{X}, \vec{Y}$ de una misma distribución con matriz de covarianzas Σ, se define la disimilaridad de Mahalanobis como:

$$d(\vec{X}, \vec{Y}) = \sqrt{(\vec{X} - \vec{Y})^t \Sigma^{-1} (\vec{X} - \vec{Y})}$$

Se observa que esta medida proviene de la distancia euclídea, pues si la matriz de covarianzas es la identidad, entonces la distancia de Mahalanobis es la distancia euclídea.
Si la matriz de covarianzas es diagonal entonces esta distancia se reduce al caso de la distancia euclídea estándar.

3.3.3 Producto escalar

Si usamos como función de energía la función proveniente del exponente del kernel gaussiano, podemos escribir $e(\mathbf{q}, \mathbf{k}) = -\frac{1}{2}||\mathbf{q} - \mathbf{k}_i||^2$, que desarrollando queda:

$$e(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^t\mathbf{k}_i - ||\mathbf{q}||^2 - ||\mathbf{k}_i||^2$$

Se observa que el segundo término solo depende de $\mathbf{q}$. Como consecuencia podemos eliminarlo. El último término no es más que la norma al cuadrado de $\mathbf{k}$. Cuando las *keys* son obtenidas mediante capas de normalización, su norma va a estar acotada o incluso va a ser constante. Por tanto también podemos despreciar ese término, quedando simplemente el producto escalar entre los dos vectores. Para dimensiones pequeñas de los datos no sería una mala opción usar el producto escalar, sin embargo, en la realidad es más común usar el producto escalar con una pequeña variación. Normalmente los datos tienen gran dimensión. Por tanto el producto escalar de dos vectores daría valores muy grandes. Debido a que debemos aplicar la función softmax, puede que vayamos a valores donde el gradiente de esta función tome valores muy pequeños. Es por eso por lo que definimos el Producto Escalar Rectificado, [23, Sección 11.3].

Definición 3.5 (Producto Escalar Rectificado).

$$e(\boldsymbol{q}, \boldsymbol{k}_i) = \frac{\boldsymbol{q}^t\boldsymbol{k}_i}{\sqrt{d_k}}$$

donde d_k es la dimensión de los vectores $\mathbf{k}$, (que coincide con la de los vectores $\boldsymbol{q}$).

Dividir por el factor d_k hace que el producto escalar no crezca excesivamente. Está función junto con la función softmax es la que se usa en el conocido artículo *Attention is all you need* [22]

3.3.4 Atención aditiva

Una nueva idea de medir la similitud entre dos vectores dados es usar una red neuronal con una sola capa oculta [23, Sección 11.3.4]. Entonces la función de atención aditiva viene definida por :

$$e(\mathbf{q}, \mathbf{k}_i) = w_v^t \tanh\left(W_q\mathbf{q} + W_k\mathbf{k}\right)$$

donde $w_v \in \mathbb{R}^h$,$W_q \in \mathbb{R}^{h\times q}$,$W_k \in \mathbb{R}^{h\times k}$ son parámetros a aprender y la función tanh sería la función de activación.

3.4 Tipos de atención

Con esto quedan explicadas las bases del mecanismo de atención. Sin embargo, hay varias formas de aplicarlo para que encaje en los distintos modelos o simplemente para que sea más rápido a la hora de la computación.

3.4.1 Batch Matrix

Si trabajamos con un texto, el número de palabras es muy alto. Por tanto, los *queries*, *keys* y *values* asociados a ellas se disponen en forma de matrices. Supongamos que tenemos tres matrices, $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, $V \in \mathbb{R}^{m \times d_v}$ que representan respectivamente los vectores *queries*, *keys* y *values*. Entonces si queremos calcular la atención utilizando el producto escalar basta multiplicar las matrices de la forma correcta:

$$\alpha(Q, K) = QK^t$$

La matriz Q contiene en cada fila un vector $\mathbf{q}$. Análogamente con las matrices K y V. Entonces si hacemos la traspuesta de K, tendremos en cada columna un vector $\mathbf{k}$. Haciendo el producto matricial obtenemos una matriz $n \times m$ donde en la fila n se encuentran los pesos de atención del query n-ésimo con respecto a todos los keys. Si ahora queremos hacer la suma ponderada de los pesos de atención con los vectores values, basta multiplicar la matriz obtenida por la matriz V.

Si las dimensiones de las matrices son muy altas, dividimos las matrices en varias matrices más pequeñas, y a esto se le denominan batches. De esta forma $Q = \left[Q_1 | Q_2 | \dots | Q_a\right]$ donde cada $Q_i \in \mathbb{R}^{n \times d_k} \forall i \in \{1, ..., a\}$. Por tanto ahora Q viene representado por un tensor de rango 3 con dimensión $\left[a, n, d_k\right]$. Lo mismo podemos hacer con las keys y los values, pero hay que tener en cuenta que el número de batches debe ser igual en los tres grupos.

3.4.2 Atención enmascarada

Si estamos trabajando con secuencias de texto es habitual encontrar símbolos que indican el final o el principio de una frase o encontrar secuencias de texto con muchos símbolos en blanco. Esto es el paso común en la preparación del texto para que todas las secuencias tengan la misma longitud. Por ejemplo,

```
Tom has been watching TV all day
I like coffee <blank> <blank> <blank> <blank>
Hello Word <blank> <blank> <blank> <blank> <blank>
```

Estos símbolos representan tokens del texto que quizás no influyen en el significado del texto, por lo que no se deberían tener en cuenta en el proceso de atención. Otras veces nos puede interesar no calcular los pesos de atención asociados a ciertas palabras por alguna razón (esto pasa en los Transformers y será explicado más adelante). Para resolver estos problemas usamos la atención enmascarada (*masked attention*) [23, Sección 11.3.2.1].

La atención enmascarada logra que ciertos tokens no influyan en el mecanismo de atención. Para ello sus vectores values asociados son 0 y los pesos de atención de estos tokens respecto de otros son números muy negativos, es decir, del orden de -10^6. De esta forma la función softmax se anula sobre valores grandes muy negativos [23, Sección 11.3].

4 Modelos que usan el mecanismo de atención

Una vez comprendido qué es el mecanismo de atención pasaremos a ver cómo los modelos usan esta herramienta para mejorar el rendimiento en algunas tareas.

4.1 El modelo de Bahdanau

En 2015, Bahdanau propone en [2] un modelo en el que usa el mecanismo de atención para realizar la tarea de traducción automática. El objetivo del artículo citado era mejorar los modelos existentes en tal tarea.

En [2] se explica que la tarea de traducción se puede ver de forma estadística. Denotaremos como x las oraciones de entrada y cómo y, las oraciones objetivo; es decir, la oración que queremos que sea traducción de x. Lo que queremos es maximizar la probabilidad condicionada de y a x. Buscamos por tanto $\arg\max_y P(y|x)$. Para calcular esta probabilidad se usa un modelo de redes recurrentes con estructura de encoder-decoder. En este tipo de estructuras el encoder codifica un dato de entrada, esta codificación pasa al decoder, y este produce una salida. Si tenemos una oración de entrada $x = (x_1, \dots, x_{T_x})$, el encoder la transforma en un vector c, que contiene información sobre la oración de entrada. Se denomina vector de contexto. El encoder en este caso es una red recurrente de tal forma que el estado oculto en la etapa t es $h_t = f(x_t, h_{t-1})$. c será el resultado de aplicar una función no lineal a los distintos estados ocultos de la red neuronal:

$$c = q(h_1, \dots, h_{T_x})$$

Hasta ese momento se usaba cómo f un red LSTM y como q la función $q(h_1, \dots, h_{T_x}) = h_{T_x}$.

Por otro lado, el decoder toma como entrada el vector c y las palabras predichas anteriormente, i.e, $\{y_1, \cdots, y_{t-1}\}$. Con esta información intenta predecir la traducción de la siguiente palabra. Para ello descompone la probabilidad de y, que es la oración objetivo, en el producto de las probabilidades condicionadas:

$$P(y) = \prod_{t=1}^{T} P(y_t|\{y_1, \dots, y_{t-1}\}, c)$$

donde $y = (y_1, \dots, y_T)$

Esta probabilidad se calcula con la ayuda de una red recurrente, que calcula las probabilidades marginales. De esta forma $P(y_t|\{y_1, \dots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$ donde s_t es el estado oculto en la etapa t de tal RNN y g es una función no lineal.

Este modelo es resumido brevemente en el artículo ya citado de Bahdanau [2], pero el modelo se presenta en un paper anterior cuyo autor es Sutskever [19]. Este modelo es importante para entender como funciona la estructura creada por Bahdanau.

En [2] se propone calcular la probabilidad anterior de una forma similar, pero con un sutil cambio:

$$P(y_t|\{y_1, \dots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c_i)$$

Ahora c_i es diferente para cada i,es decir, para cada etapa. Este vector dependerá de los estados ocultos de una red recurrente $(h_1, \dots, h_{T_x})$ que serán definidos posteriormente. Por ahora, nos basta saber que cada h_i contiene información de la palabra en la posición i de la oración de entrada, por tanto esta información también depende de las palabras que rodean a x_i. Entonces el vector c_i se define como una suma ponderada de estas anotaciones h_i:

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

Esta expresión resulta familiar. Además los α_{ij} son el resultado de aplicar la función softmax a los vectores $e_{ij} = a(s_{i-1}, h_j)$, donde $a(\cdot, \cdot)$ es una red neuronal feedforward que mide la relación entre el estado oculto de la RNN, s_{i-1}, y la anotación, h_j, de la oración de entrada.

En otras palabras, Bahdanau ha aplicado el mecanismo de atención con la función de puntuación antes definida como atención aditiva donde los queries son ahora los

estados ocultos de una RNN y los keys coinciden con los values y son las denominadas puntuaciones h_j.

Cómo las anotaciones h_j dependen de la palabra precedida y la palabra siguiente a x_j, Bahdanau propone usar una red recurrente bidireccional. Esta está formada por dos RNN. Una lee la palabra hacia adelante y otra hacia atrás. Cada red da lugar a unos estados ocultos que denotaremos como $\vec{h}_1, \dots, \vec{h}_t$ a los de la RNN que lee la palabra hacia adelante, y como $\overleftarrow{h}_1, \dots, \overleftarrow{h}_t$ a los estados ocultos de la segunda RNN. La anotación final será la concatenación de las anteriores: $h_j = \left[\vec{h}_j^t | \overleftarrow{h}_j^t\right]$.

Aunque este modelo produjo una mejora en la tarea de traducción, la llegada de los Transformers han tenido muchísimo éxito, pues son los modelos que se encuentran detrás del chatGPT, desarrollado por la empresa *OpenAI* que tanto da qué hablar en la actualidad. Veamos qué son los Transformers.

4.2 El modelo Transformer

Los Transformers son una estructura de redes neuronales que nacieron en el artículo "***Attention is all you need***", [22], con la tarea de resolver un problema de traducción de texto. Con el paso del tiempo, varios profesionales de la materia han aplicado esta estructura a diferentes tareas. La estructura de los Transformes se representa en el gráfico 4.1. A continuación se desarrollará cada elemento de esta estructura.

4.2.1 Embedding

Como se va a trabajar con texto, es necesario crear una estructura matemática que represente el texto para que la red neuronal pueda entenderlo. Un embedding es una función $W : palabras \to \mathbb{R}^n$ que transforma las palabras en un espacio vectorial de dimensión finita. La dimensión dependerá del conjunto de palabras y de la tarea a realizar [14]. En el caso del artículo *Attention is all you need*, la dimensión del espacio es $d_{model} = 512$. Al principio los vectores toman posiciones aleatorias. La idea es que con un proceso de entrenamiento, las palabras vayan tomando posiciones significativas, es decir, vectores que estén a poca distancia representarán palabras similares. La similitud dependerán de la tarea y del contexto. Por ejemplo, si queremos representar

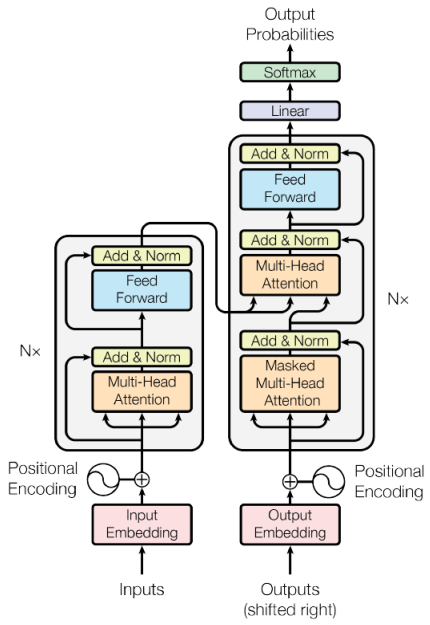


Figura 4.1: Estructura del Transformer [22]

la palabra "aprendizaje" no tendrá la misma representación en un contexto de *Machine Learning* que en el contexto de Psicología. Además, en el espacio de embedding hay vectores asociados al género o al plural. De esta forma, si tenemos codificada la palabra "rey" y le sumamos el vector "femenino" nos llevará a un vector que represente la palabra "reina". Podemos obtener embeddings mediante una capa de embedding, la cual está implementada en librerías donde se trabaja con redes neuronales (por ejemplo Keras en python), o bien, escoger embeddings ya entrenados para cierta tarea. La capa de embedding funciona como un diccionario: Dado una palabra le hace corresponder un vector. Los valores de la capa son los valores de ese diccionario, que a través del algoritmo de backpropagation, se van modificando para crear la mejor representación [3, Pág 331].

4.2.2 Positional encoding

Ya tenemos las palabras codificadas de tal forma que el Transformer las pueda entender. Por tanto podemos codificar una oración. Pero hay un problema, la red neuronal no sabe en qué posición está cada palabra. La frase "El chico limpia la casa" tiene total sentido. Sin embargo, "la casa chico limpia el" carece de sentido y además tiene

la misma codificación que la frase anterior. Es por eso por lo que los creadores de esta arquitectura, Vawnasi et al. [22], añaden al vector del embedding un nuevo vector, el vector de **Posititonal Encoding**.

Este vector es diferente para cada posición. Este viene definido según una función, $f : \{1, \cdots, pos\} \to \mathbb{R}^{d_{model}}$, que para cada posición $t \in \{1, \cdots, pos\}$, nos da el vector que codifica tal posición. Según la definición propuesta es importante que la dimensión de los vectores del embedding sea par, pues las componentes de los vectores que codifican la posición se definen por parejas de funciones trigonométricas de la siguiente forma, [8]:

$$f^{(i)}(t) = \begin{cases} \sin(\omega_k \cdot t) \; si \; i = 2k \\ \cos(\omega_k \cdot t) \; si \; i = 2k+1 \end{cases}$$

donde $\omega_k = \frac{1}{10000^{\frac{2k}{d}}} = \left[\frac{1}{10000^{\frac{2}{d}}}\right]^k$.

Según esta definición se crea una progresión geométrica desde 2π a $10000 \cdot 2\pi$. Además, si tenemos fijada una posición, $pos = \delta$, podemos calcular el positional encoding de la posición $\delta + t$ a través de una transformación lineal de $f(t)$, como podemos ver en [8],[23, Sección 11.6.3].

Dicho de otro modo queremos probar que

$$\exists M^{(\delta)} \in \mathbb{R}^{2\times 2} \; t.q \; M^{(\delta)} \begin{bmatrix} \sin(\omega_j t) \\ \cos(\omega_j t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_j(\delta + t)) \\ \cos(\omega_j(\delta + t)) \end{bmatrix}$$

Si tenemos esto para cada par $\sin - \cos$, entonces tenemos que podemos calcular el vector que codifica la posición $\delta + t$ a partir del vector que codifica la posición t. La construcción se haría multiplicando la matriz siguiente por el vector que codifica la posición t

$$T^{\delta} = \begin{bmatrix} M_1 & 0 & \cdots & 0 \\ 0 & M_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & M_{\frac{dmodel}{2}} \end{bmatrix} \tag{4.1}$$

donde los 0 de la matriz representan matrices cuadradas nulas. De esta forma se tendría que $T^{\delta} f(t) = f(t + \delta)$

Veamos cómo está formada la matriz M. Sea $M = \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix}$.

Entonces

$$\begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \begin{bmatrix} \sin(\omega_j t) \\ \cos(\omega_j t) \end{bmatrix} = \begin{bmatrix} u_1 \sin(\omega_j t) + u_2 \cos(\omega_j t) \\ v_1 \sin(\omega_j t) + v_2 \cos(\omega_j t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_j(\delta + t)) \\ \cos(\omega_j(\delta + t)) \end{bmatrix} \tag{4.2}$$

Podemos desarrollar el vector de la derecha usando el seno de la suma y el coseno de la suma, que vienen dados por:

$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

De esta forma,

$$\begin{bmatrix} \sin(\omega_j(\delta + t)) \\ \cos(\omega_j(\delta + t)) \end{bmatrix} = \begin{bmatrix} \sin(\omega_j \delta)\cos(\omega_j t) + \cos(\omega_j \delta)\sin(\omega_j t) \\ \cos(\omega_j \delta)\cos(\omega_j t) - \sin(\omega_j \delta)\sin(\omega_j t) \end{bmatrix} \tag{4.3}$$

Sustituyendo 4.3 en 4.2, podemos despejar los valores de u_1, u_2, v_1, v_2:

$$\begin{bmatrix} u_1 \sin(\omega_j t) + u_2 \cos(\omega_j t) \\ v_1 \sin(\omega_j t) + v_2 \cos(\omega_j t) \end{bmatrix} = \begin{bmatrix} \sin(\omega_j \delta)\cos(\omega_j t) + \cos(\omega_j \delta)\sin(\omega_j t) \\ \cos(\omega_j \delta)\cos(\omega_j t) - \sin(\omega_j \delta)\sin(\omega_j t) \end{bmatrix} \tag{4.4}$$

Lo que nos lleva a que

$$M = \begin{bmatrix} \cos(\omega_j \delta) & \sin(\omega_j \delta) \\ -\sin(\omega_j \delta) & \cos(\omega_j \delta) \end{bmatrix}$$

La intuición sobre la elección de la codificación mediante senos y cosenos viene de la idea de enumerar las posiciones con números binarios. Si miramos los primeros 11 números en formato binario en la tabla 4.1, podemos observar que en la columna azul, el 0 y el 1 se cambian en cada posición; en la columna roja, cada dos posiciones, en la columna verde cada cuatro posiciones y así sucesivamente. Podemos interpretar esta rotación de ceros y unos como una onda discreta. Una codificación de esta forma implica una gran cantidad de espacio, por tanto, la mejor opción es escoger una versión continua de esta codificación,es decir, una codificación en forma de senos y cosenos [8].

En resumen, el positional encoding es una pequeña componente del Transformer que logra que las palabras tengan un orden. Es más en el artículo original [22] establecen la idea en un pequeño párrafo. Sin embargo, he querido profundizar más ya que es una simple idea que tiene detrás mucho contenido matemático. Con el embedding y el positional encoding tenemos codificadas las palabras. Cada palabra tendrá asociada tres vectores, $\mathbf{q} \in \mathbb{R}^{d_k}$,$\mathbf{k} \in \mathbb{R}^{d_k}$ y $\mathbf{v} \in \mathbb{R}^{d_v}$. Vistas las primeras componentes del Transformer pasamos a ver el encoder y el decoder, elementos muy importantes de este.

0:0000	6:0110
1:0001	7:0111
2:0010	8:1000
3:0011	9:1001
4:0100	10:1010
5:0101	11:1011

Cuadro 4.1: Representación de las posiciones en número binario

4.2.3 Estructura del Transformer

El Transformer tiene una estructura de encoder-decoder parecida a la del modelo de Bahdanau (Sección 3.6). En este caso el encoder y el decoder están formados por varias capas. En el artículo original, ambos están formados por 6 capas. Estas están conectadas entre sí, tanto en el encoder como en el decoder, a diferencia de que la última capa del encoder se conecta con la primera capa del decoder. Si estamos con una tarea de traducción, el encoder recibe como entrada las frases que queremos traducir, mientras que el decoder, recibe como entrada la traducción de estas. A continuación se estudiará como están formados estos elementos del Transformer.

Encoder

Cada capa del encoder está formada por dos subcapas: un mecanismo de atención y una red neuronal feed forward. El mecanismo de atención recibe como entrada los queries, keys y values procedentes de los vectores generados por el embedding más el positional encoding sobre los valores de entrada. Sin embargo los autores proponen una nueva forma de aplicar la atención, lo que se conoce como *Multi-Head attention*. En vez de aplicar directamente la atención sobre los vectores queries,keys y values de dimensión 512, se aplican varias proyecciones lineales y se aplica el mecanismo de atención sobre estas en paralelo. Estas transformaciones no son más que 4 matrices con parámetros que son aprendidas por la red. Hay 3 matrices asociadas a las matrices Q,K,V (matrices de los queries, keys y values) respectivamente y una cuarta para volver a la dimensión original. Supongamos que hacemos h proyecciones, entonces:

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v} i \in \{1, .., h\}$$

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

Después de realizar estas transformaciones, se concatenan y se multiplican por la matriz W^o para volver a la dimensión del modelo. Sea

$$h_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \in \mathbb{R}^{d_{model} \times d_v}$$

Entonces $H = \left[h_1 | \cdots | h_h\right] \in \mathbb{R}^{d_{model} \times hd_v}$ y el resultado de hacer *Multi-Head attention* es [22]

$$\text{MultiHead}(Q, K, V) = HW^o \in \mathbb{R}^{d_{model} \times d_{model}}$$

En el artículo se habla de un tipo de atención hasta ahora no citada denominada *self-attention*. No obstante, este tipo de atención no tiene nada nuevo. Simplemente se le da este nombre porque se aplica la atención con los queries, keys y values procedentes de la capa anterior. De esta forma, cada token del encoder puede prestar atención con cada uno de los tokens de la capa anterior[23, Sección 11.6]. En el artículo [22] se aplica como función de puntuación el producto escalar rectificado por un factor que es $\frac{1}{\sqrt{d_k}}$, y a este se le aplica la función softmax. De esta forma se tiene:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V$$

El mecanismo de atención junto con el proceso de multi-head forman la primera capa del encoder, que da como salida una matriz de dimensión $d_{model} \times d_{model}$. La segunda subcapa es una red feed-forward formada por dos capas con la función de activación *ReLu* en la primera capa:

$$\text{FFN}(x) = ReLu(W_1 + b_1) \cdot W_2 + b_2$$

Entre estas capas se añaden conexiones residuales y capas de normalización, que se explicarán más adelante.

Veamos ahora la estructura del decoder.

Decoder

El decoder tiene una estructura parecida a la del encoder, sin embargo tiene 3 subcapas en vez de 2. Esta tercera capa es un proceso multihead pero esta vez con atención enmascarada (*masked attention*). Los valores que toma el decoder son las salidas esperadas de nuestro conjunto de datos. Por ejemplo, en la tarea de traducción

serían las frases traducidas. Como es un modelo secuencial, queremos que el decoder no anticipe información futura, es decir que la palabra generada por el decoder solo tenga en cuenta las palabras que ya han sido generadas anteriormente. Por tanto aplicamos atención enmascarada a los valores que nos lleven a conexiones prohibidas. Si no tenemos esto en cuenta, el modelo se limitará a copiar la información de los datos de entrada objetivos (más comúnmente llamados target), provocando así que a la hora de traducir un texto nuevo no pueda hacerlo, pues solo sabe la traducción de los datos de entrenamiento[3, Pág 360],[22]. En cuanto a la segunda subcapa, también es una capa de *multihead attention*, aunque aquí los queries provienen de la subcapa precedente del decoder, y los keys y values de la salida del encoder. Este tipo de atención se conoce como atención cruzada o atención encoder-decoder. La tercera subcapa es, al igual que en el decoder, una red feed-forward. Entre todas estas subcapas también se añaden conexiones residuales y capas de normalización, al igual que en el encoder. La salida del decoder pasa a través de una capa lineal y una función softmax que dará la probabilidad de que cierta palabra sea la traducción de la palabra de entrada en ese paso.

Capas de normalización y conexiones residuales

Estos dos elementos tienen la función de que el entrenamiento sea más rápido. Empecemos con las conexiones residuales.

Las **conexiones residuales** se crean para solucionar un problema que se tiene cuando las redes neuronales son muy profundas. El problema es que cuanto mayor número de capas, puede ser que se vaya acumulando los errores producidos en cada capa, y esto produce que el algoritmo del gradiente deje de ser efectivo. Por tanto, en la práctica los errores de entrenamiento y los errores sobre la muestra test pueden ser mayores en redes más profundas. La idea para solventar este problema es dejar pasar la información de una capa a otra sin que esta sea procesada por capas intermedias. Esta idea fue desarrollada de manera teórica en el artículo [6]. Sin embargo, me resulta interesante la metáfora que hace Chollet en [3, Pág 253], y es que lo compara con el juego del teléfono, cuyo objetivo es que el mensaje llegue desde el emisor al receptor pasando por varias personas. En la práctica, el mensaje emitido nunca llega al receptor, y si lo hace es de forma muy distinta, provocando una pérdida de información. La analogía con las redes neuronales sería cambiar el mensaje, por los datos y los participantes, por las capas de la red. De igual forma que el problema queda resuelto si el emisor hace llegar directamente la información al receptor en el juego del teléfono, en el ámbito de las redes neuronales, el problema quedaría resuelto si se

pasa la información directamente de una capa a otra. De forma matemática, en el modelo Transformer se entendería que la información de entrada en la segunda o tercera subcapa (si estamos en el decoder), sería $x + \text{sublayer}(x)$.

La **capa de normalización** tiene como antecedente otro tipo de normalización llamada *batch normalization* [7] . Este se aplicó en un principio en redes feed forward y sirve para que las neuronas de estas redes puedan trabajar más rápido. La distribución de los datos va sufriendo cambios en cada capa. Este cambio produce el problema de que las capas se tienen que adaptar a la nueva distribución de los datos continuamente. Los investigadores se dieron cuenta que esto producía un entrenamiento muy lento, incluso a veces, no se conseguía minimizar la función de coste lo suficiente. Por eso en [7], se crea una nueva capa, denotada $\boldsymbol{BN}(x)$ (las iniciales de *batch normalization*), que se aplica entre las capas de la red. Esta se define como sigue: Sea $\boldsymbol{B}$ un mini-batch y $x \in \boldsymbol{B}$ la entrada de la capa de normalización. Entonces

$$\boldsymbol{BN}(x)^{(k)} = \gamma^{(k)} \cdot \frac{x^k - \mu_{\boldsymbol{B}}^{(k)}}{\sigma_{\boldsymbol{B}}^{(k)}} + \beta^{(k)}$$

donde $\mu_{\boldsymbol{B}}$ es la media muestral, $\sigma_{\boldsymbol{B}}$ es la desviación típica muestral y γ, β son parámetros a aprender. Se observa que si es mejor no normalizar los datos en el modelo, entonces sustituyendo γ por $\sigma_{\boldsymbol{B}}$ y β por $\mu_{\boldsymbol{B}}$, recuperamos los datos iniciales, [23, Sección 8.5.1][7].

En un modelo secuencial no sirve esta capa porque la media y la varianza muestral dependen del tamaño del batch. Es por eso que en [1] proponen una generalización a la capa de *batch normalization*, llamada *layer normalization* o capa de normalización. Sea $\boldsymbol{H}$ el número de tokens de la secuencia de entrada, y sea a^l el vector que representa el vector de salida de la capa anterior. Entonces se define μ^l y σ^l como sigue:

$$\mu^l = \frac{1}{H} \sum_{i=1}^{H} a_i^l$$

$$\sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^{H} (a_i^l - \mu^l)^2}$$

La capa de normalización sigue la misma idea que la capa de normalización batch pero la media y la varianza usadas para la tipificación son las descritas anteriormente. Por tanto una capa de normalización viene dada por la expresión

$$\frac{g}{\sigma^l} \odot (a_i^l - \mu^l) + b$$

donde g y b son parámetros a aprender por la red.

5 Experimentación

Una vez explicados los aspectos matemáticos del mecanismo de atención y la estructura del Transformer, pasamos a estudiar como se implementa este modelo en Python, que es el lenguaje habitual para trabajar con redes neuronales, en particular, con los Transformers. El código escrito a continuación es una recompilación del capítulo 11 del libro de Chollet [3], que podemos encontrar en https://github.com/fchollet/deep-learning-with-python-notebooks.

La forma más habitual de trabajar en *Deep Learning* es a través de la librería de Python *Keras*, creada por François Chollet. *Keras* está basada en otra librería llamada *Tensorflow*. Esta es una herramienta de Python que permite trabajar con tensores y está especializada para ser utilizada en *Machine Learning* . En este caso, el código ha sido ejecutado en *Google Colaboratory*.

Lo primero que debemos hacer es preparar el texto para introducirlo en el Transformer. En la siguiente sección veremos como trabajar con un dataset de texto además de ver como funcionan los embeddings.

5.1 Preparación del texto

Primero debemos cargar el dataset, que en este caso se obtiene de la página de Stanford y corresponde al conjunto de datos IMBD de críticas de películas. Este conjunto de datos contiene críticas positivas y negativas.

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/
aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup
#Se obtiene los datos de la página web descrita en la línea 1.
```

```
#!rm sirve para borrar  un subdirectotio que no vamos a usar
```

El siguiente código crea un conjunto de entrenamiento y otro conjunto de validación. En ambos conjuntos tenemos tanto críticas positivas como negativas. El conjunto de validación contiene un 20 % de los archivos totales. Hay 25000 archivos, 20000 formarán el conjunto de entrenamiento y 5000 el de validación. Se fija una semilla para que siempre trabajemos con el mismo conjunto de datos, ya que son escogidos de forma aleatoria.

Listing 5.1 : Conjuntos de validación y entrenamiento

```
import os, pathlib, shutil, random
from tensorflow import keras
batch_size = 32
base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
text_only_train_ds = train_ds.map(lambda x, y: x)

Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
```

Con el texto ya preparado podemos usar diferentes formas de codificación. En nuestro caso será mediante capas de embedding. Para ello primero necesitamos pasar

el texto a una secuencia de números enteros. Para ello se usa la capa `TextVectorization`.

Listing 5.2 : Codificación a través de la capa TextVectorization

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

Los conjuntos creados aquí sirven para entrenar a un modelo. `int_train_ds` será el conjunto que usamos para el entrenamiento, `int_val_ds` el conjunto de validación y `int_test_ds` el conjunto para medir el error del modelo. Cabe destacar que los conjuntos creados en el listing 5.1 son usados para definir los conjuntos del listing 5.2

Para crear un embedding, *Keras* tiene implementada una capa de embedding:

Listing 5.3 : Capa de Embedding

```
embeding_layer=layers.Embedding(input_dim=max_tokens,
output_dim=256)
#Input_dim = número de posibles tokens
#output_dim = dimensión del embedding.
```

El primer modelo que se presenta está formado por una capa de embedding, una red LSTM bidireccional, y una capa densa (una capa feed-forward) que da la salida final. Su implementación en Keras es la siguiente:

Listing 5.4 : Modelo que usa una capa de embedding desde cero

```
inputs = keras.Input(shape=(None,), dtype="int64")
```

```
embedded = layers.Embedding(input_dim=max_tokens,
                output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

#A partir de aquí empieza el entrenamiento
callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,

callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.keras")
print(f"Test␣acc:␣{model.evaluate(int_test_ds)[1]:.3f}")
```

En este caso, como estamos usando redes recurrentes bidireccionales, las secuencias de texto tienen todas la misma longitud. Esto se consigue rellenado con ceros aquellas frases que tengan menor longitud que la establecida y truncando aquellas que superen tal longitud. Como consecuencia, es poco eficiente que la red tome estos últimos valores de relleno en el entrenamiento. La solucióon se hace mediante un proceso de enmascaramiento (*masking*). Para ello se crea un tensor de ceros y unos, de dimensiones (tamaño del batch, longitud de la secuencia). Si en el paso de tiempo t se encuentra un cero, esto quiere decir que no es necesario ejecutar ese token. Para aplicar esta herramienta, tan solo hay que indicar en la capa de embedding `mask_zero=true`. Cuando se implemente la atención enmascarada correspondiente al decoder se hará un proceso parecido [3, Pág. 332-333].

Listing 5.5 : Capa de embedding que usa *masking*

```
    layers.Embedding(
    input_dim=max_tokens, output_dim=256, mask_zero=True)
```

El modelo descrito en el listing 5.4 entrena una capa de embedding desde 0. Por tanto es un modelo lento de entrenar. Es habitual usar embedding ya entrenados. En [3, Cap. 11] se propone usar una capa ya entrenada llamada GloVe (*Global Vector for Word Representations*). Para usarlo, primero debemos descargarlo.

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip
```

A continuación debemos descomprimir el archivo y después enumerar las palabras que este contiene.

Listing 5.6 : Preparación del archivo *GloVe*

```
import numpy as np
path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep="␣")
        embeddings_index[word] = coefs

print(f"Found␣{len(embeddings_index)}␣word␣vectors.")

Found 400000 word vectors.
```

El objetivo es poder usar los vectores de GloVe en una capa de embedding. Para ello primero debemos construir una matriz que contenga en cada fila el vector asociado a una palabra y que tenga tantas columnas como la dimensión de los vectores del embedding. Esto se consigue mediante el siguiente código:

Listing 5.7 : Definición de una matriz para el embedding

```
embedding_dim = 100

vocabulary = text_vectorization.get_vocabulary()
word_index = dict(zip(vocabulary, range(len(vocabulary))))

embedding_matrix = np.zeros((max_tokens, embedding_dim))
for word, i in word_index.items():
    if i < max_tokens:
        embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Esta matriz está lista para ser la entrada de una capa de embedding. Como ya tenemos los vectores deseados, no queremos que la capa de embedding modifique tales vectores. Por tanto, podemos indicar `trainable=False` para que la capa de embed-

ding no realice el proceso de entrenamiento. La capa de embedding que usaremos en el modelo viene dada por:

Listing 5.8 : Implementación de la capa de embedding

```
embedding_layer = layers.Embedding(
    max_tokens,
    embedding_dim,
    embeddings_initializer=
    keras.initializers.Constant(embedding_matrix),
    trainable=False,
    mask_zero=True,
)
```

Cabe destacar que la matriz de embedding definida en el listing 5.7 es la que se usa en esta capa de embedding. Ya podemos definir un modelo que use esta capa:

Listing 5.9 : Modelo que usa un embedding entrenado previamente

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = embedding_layer(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
keras.callbacks.ModelCheckpoint(
"glove_embeddings_sequence_model.keras",
                save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,

callbacks=callbacks)
model =

keras.models.load_model("glove_embeddings_sequence_model.keras")
print(f"Test␣acc:␣{model.evaluate(int_test_ds)[1]:.3f}")
```

La salida del listing 5.9 es:

Listing 5.10 : Entrenamiento del modelo

```
    Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, None)]            0

 embedding_3 (Embedding)     (None, None, 100)         2000000

 bidirectional_3 (Bidirectio  (None, 64)               34048
 nal)

 dropout_3 (Dropout)         (None, 64)                0

 dense_3 (Dense)             (None, 1)                 65

=================================================================
Total params: 2,034,113
Trainable params: 34,113
Non-trainable params: 2,000,000
_________________________________________________________________
Epoch 5/10
625/625 [==============================] - 514s 822ms/step -
loss: 0.3638 - accuracy: 0.8421 - val_loss: 0.3402 -
val_accuracy: 0.8522
Epoch 6/10
625/625 [==============================] - 515s 824ms/step -
loss: 0.3427 - accuracy: 0.8540 - val_loss: 0.3167 -
val_accuracy: 0.8708
Epoch 7/10
625/625 [==============================] - 511s 818ms/step -
loss: 0.3198 - accuracy: 0.8680 - val_loss: 0.3050 -
val_accuracy: 0.8724
Epoch 8/10
625/625 [==============================] - 508s 812ms/step -
loss: 0.3059 - accuracy: 0.8719 - val_loss: 0.2992 -
val_accuracy: 0.8748
Epoch 9/10
625/625 [==============================] - 505s 808ms/step -
loss: 0.2901 - accuracy: 0.8801 - val_loss: 0.3058 -
val_accuracy: 0.8732
Epoch 10/10
625/625 [==============================] - 515s 824ms/step -
loss: 0.2717 - accuracy: 0.8887 - val_loss: 0.2938 -
val_accuracy: 0.8802
```

```
782/782 [==============================] - 150s 187ms/step -
loss: 0.3040 - accuracy: 0.8704
Test acc: 0.870
```

Ya hemos visto como crear capas de embedding desde cero y como usar capas de embedding ya entrenadas como *GloVe*. Estamos listos para pasar al modelo Transformer.

5.2 Implementación del Modelo Transformer

Empezaremos por preparar los datos que sirven para entrenar al Transformer. En este ejemplo vamos a descargar datos que contienen oraciones en inglés junto con su traducción al español.

Listing 5.11 : Carga del dataset

```
!wget
http://storage.googleapis.com/download.tensorflow.org/data/
spa-eng.zip
!unzip -q spa-eng.zip
```

En este dataset, cada línea contiene las dos oraciones en inglés y en español separadas por un caracter. El siguiente código trata de separar las oraciones del dataset, de forma que se identifiquen mejor las frases en inglés y en español:

Listing 5.12 : preparación del dataset

```
    text_file = "spa-eng/spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

Si queremos ver cómo se disponen las oraciones del dataset, podemos seleccionar una línea al azar:

Listing 5.13 : Dato aleatorio

```
import random
```

```
print(random.choice(text_pairs))
#Salida:
("I␣just␣don't␣want␣to␣talk␣to␣you.",
'[start]␣Sencillamente␣no␣quiero␣hablar␣contigo.␣[end]')
```

Una vez comprendida la forma de nuestro dataset pasamos a crear conjuntos de entrenamiento, validación y test:

Listing 5.14 : Creación de conjuntos de validación entrenamiento y test

```
import random
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples
+ num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

Ahora se lleva a cabo un proceso de tokenización, es decir, se eliminan los acentos y algunos signos de puntuación como ¿ en el caso del español. Para cada idioma usaremos una capa de `TextVectorization`:

Listing 5.15 : Vectorizacón del texto

```
from tensorflow.keras.layers import TextVectorization
import tensorflow as tf
import string
import re

strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")

#Solo se eligen las primeras 15000 palabras de cada idioma
#y las frases que tengan menos de 20 palabras.
vocab_size = 15000
sequence_length = 20
#Esta es la que se aplica a las frases en inglés
source_vectorization = TextVectorization(
    max_tokens=vocab_size,
```

```
    output_mode="int",
    output_sequence_length=sequence_length,
)
#Esta es la que se aplica a las frases en inglés
target_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

Finalmente, convertimos el texto a tensores de la librería Tensorflow:

Listing 5.16 : Transformación a vectores

```
batch_size = 64

def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({
        "english": eng,
        "spanish": spa[:, :-1],
    }, spa[:, 1:])

def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices(
    (eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset)
    return dataset.shuffle(2048).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

Podemos ver la forma del conjunto de datos:

```
for inputs, targets in train_ds.take(1):
    print(f"inputs['english'].shape: {inputs['english'].shape}")
```

```
    print(f"inputs['spanish'].shape: {inputs['spanish'].shape}")
    print(f"targets.shape: {targets.shape}")

#La salida es:
inputs['english'].shape: (64, 20)
inputs['spanish'].shape: (64, 20)
targets.shape: (64, 20)
```

A continuación se definirán las estructuras principales del Transformer: el encoder y el decoder. Para ello primero debemos implementar el mecanismo de atención. La librería *Keras*, dispone de una capa llamada `MultiHeadAttention` [3, Pág 339].

```
num_heads = 4
embed_dim = 256
mhs_layer = MultiheadAttention(num_heads, key_dim = embed_dim)
outputs = mha_layer(inputs,inputs,inputs)
```

Veamos cómo se implementa el encoder:

Listing 5.17 : El encoder del Transformer

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        #Define la atención Multi-head
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
           #Esta es la segunda subcapa del encoder
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        #Capas de normalización
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
```

```
        attention_output = self.attention(
            inputs, inputs, attention_mask=mask)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

En el libro [3], Chollet no usa el *Positional Encoding* descrito en el trabajo. Opta por usar otro método para codificar la posición. En concreto, usa una capa de embedding para producir vectores que lleven asociada una posición, y después suma estos a los vectores originales. Esta es la propuesta de Chollet.

Listing 5.18 : Codificación de la posición

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim,
    **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
```

```
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

No obstante, he querido investigar la forma de implementar el *Positional Encoding* que se propone en el artículo original [22]. Esta información ha sido obtenida de la página oficial de *Keras*: https://keras.io/guides/keras_nlp/getting_started/. De aquí he aprendido cómo instalar un subpaquete de *Keras* especializado en procesamiento del lenguaje natural (nlp:*Natural Language Procesing*). A través de una guía que se encuentra en la misma página web, https://keras.io/api/keras_nlp/modeling_layers/sine_ position_encoding/ he comprobado cómo se implementa el *Positional Encoding* que codifica las posiciones a través de cosenos y senos. El código viene dado por:

Listing 5.19 : Descargar la librería keras_nlp

```
!pip install -q --upgrade keras-nlp tensorflow
```

Listing 5.20 : Positional Encoding

```
import keras_nlp
import tensorflow
from tensorflow import keras
# create a simple embedding layer with sinusoidal positional
#encoding
seq_len = 100
vocab_size = 1000
embedding_dim = 32
inputs = keras.Input((seq_len,), dtype=tf.float32)
embedding = keras.layers.Embedding(
    input_dim=vocab_size, output_dim=embedding_dim
)(inputs)
positional_encoding =
keras_nlp.layers.SinePositionEncoding()(embedding)
outputs = embedding + positional_encoding
```

El Decoder del Transformer viene dado por:

Listing 5.21 : Decoder del Transformer

```
class TransformerDecoder(layers.Layer):
def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
    super().__init__(**kwargs)
```

```
    self.embed_dim = embed_dim
    self.dense_dim = dense_dim
    self.num_heads = num_heads
    self.attention_1 =
    #Primera subcapa
    layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_dim)
    #Segunda subcapa
    self.attention_2 = layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_dim)
        #Tercera subcapa
        self.dense_proj = keras.Sequential(
        [layers.Dense(dense_dim, activation="relu"),
         layers.Dense(embed_dim),])
        #Capas de normalización
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

    def get_config(self):
        config = super(TransformerDecoder, self).get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

En el decoder se necesita atención enmascarada en la primera subcapa. Para ello se define una matriz que tiene ceros en la mitad superior y unos en la mitad inferior

Listing 5.22 : Definición del enmascaramiento

```
def get_causal_attention_mask(self, inputs):
    input_shape = tf.shape(inputs)
    batch_size, sequence_length = input_shape[0], input_shape[1]
    i = tf.range(sequence_length)[:, tf.newaxis]
    j = tf.range(sequence_length)
    mask = tf.cast(i >= j, dtype="int32")
    mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1),
         tf.constant([1, 1], dtype=tf.int32)], axis=0)
    return tf.tile(mask, mult)
```

Listing 5.23 : Paso final de la definición del Decoder

```
def call(self, inputs, encoder_outputs, mask=None):
   #Aquí se usa la definición del listing anterior
    causal_mask = self.get_causal_attention_mask(inputs)
    if mask is not None:
        padding_mask = tf.cast(
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    #En la primera salida de atención se usa el casual_mask
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    attention_output_1 = self.layernorm_1(
    inputs + attention_output_1)
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    proj_output = self.dense_proj(attention_output_2)
    return self.layernorm_3(attention_output_2 + proj_output)
```

Una vez definidas las clases del Encoder y del Decoder podemos construir el modelo de Transformer final:

Listing 5.24 : Modelo de Transformer final

```
embed_dim = 256
dense_dim = 2048
num_heads = 8

encoder_inputs = keras.Input(shape=(None,), dtype="int64",
name="english")
x = PositionalEmbedding(sequence_length, vocab_size,
embed_dim)(encoder_inputs)
encoder_outputs = TransformerEncoder(embed_dim, dense_dim,
num_heads)(x)

decoder_inputs = keras.Input(shape=(None,), dtype="int64",
name="spanish")
```

```
x = PositionalEmbedding(sequence_length, vocab_size,
embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, dense_dim,
num_heads)(x, encoder_outputs)
x = layers.Dropout(0.5)(x)
decoder_outputs=layers.Dense(vocab_size, activation="softmax")(x)
transformer = keras.Model([encoder_inputs, decoder_inputs],
decoder_outputs)
```

Listing 5.25 : Entrenamiento del Transformer

```
transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

Listing 5.26 : Últimas vueltas de entrenamiento

```
Epoch 25/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6783 - accuracy: 0.7513-val_loss: 2.3739 - val_accuracy: 0.6696
Epoch 26/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6681 - accuracy: 0.7531-val_loss: 2.3970 - val_accuracy: 0.6702
Epoch 27/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6545 - accuracy: 0.7559-val_loss: 2.3876 - val_accuracy: 0.6694
Epoch 28/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6416 - accuracy: 0.7582-val_loss: 2.4626 - val_accuracy: 0.6683
Epoch 29/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6285 - accuracy: 0.7605-val_loss: 2.4464 - val_accuracy: 0.6692
Epoch 30/30
1302/1302 [==============================]-88s 68ms/step - loss:
1.6187 - accuracy: 0.7620-val_loss: 2.4466 - val_accuracy: 0.6720
<keras.callbacks.History at 0x7f9cfcc11d20>
```

Para visualizar el resultado, debemos descodificar las secuencias de texto. Esta operación se realiza mediante el siguiente código:

Listing 5.27 : Descodificaión de las oraciones

```
import numpy as np
spa_vocab = target_vectorization.get_vocabulary()
```

```
spa_index_lookup = dict(zip(range(len(spa_vocab)), spa_vocab))
max_decoded_sentence_length = 20

def decode_sequence(input_sentence):
    tokenized_input_sentence = source_vectorization(
    [input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        tokenized_target_sentence = target_vectorization(
        [decoded_sentence])[:, :-1]
        predictions = transformer(
        [tokenized_input_sentence, tokenized_target_sentence])
        sampled_token_index = np.argmax(predictions[0, i, :])
        sampled_token = spa_index_lookup[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence

test_eng_texts = [pair[0] for pair in test_pairs]
for _ in range(20):
    input_sentence = random.choice(test_eng_texts)
    print("-")
    print(input_sentence)
    print(decode_sequence(input_sentence))
```

Listing 5.28 : Resultado final

```
    -
I know you like coffee.
[start] sé que café [end]
-
I go to school at seven o'clock.
[start] yo voy a la escuela a las siete [end]
-
You speak French, right?
[start] hablas francés verdad [end]
-
I don't know when she will leave for London.
[start] no sé cuándo llegará a londres [end]
-
They were lucky.
[start] eran suerte [end]
-
Tom has been watching TV all day.
[start] tom ha estado viendo la televisión todo el día [end]
```

Cómo podemos ver, hay algunos resultados buenos y otros no muy acertados. La precisión del modelo es de 0.76 %. Se puede mejorar bastante, pero hay que tener en cuenta que es un ejemplo académico.

6 Conclusiones

Tareas como la traducción automática o la generación de texto han sido un problema hace unos años atrás. Me parece interesante que a través de funciones matemáticas que midan la cercanía de las palabras entre sí haya mejorado tanto los resultados en estas tareas. Más aún, es impresionante que esta idea no solo se pueda aplicar a palabras, sino que también es de gran utilidad a la hora de trabajar con imágenes. Al fin y al cabo, medir distancias es lo primero que se enseña en un curso de topología básica.

Gracias a la base matemática obtenida en los estudios del grado, he podido aprender y disfrutar del inmenso mundo de la Inteligencia Artificial. Una vez más las matemáticas están detrás de las nuevas tecnologías. En un futuro, me gustaría seguir aprendiendo en el área del procesamiento del lenguaje natural. Quiero descubrir las matemáticas que hay detrás de modelos específicos como BERT, Chat GPT o Microsoft Bing. Además me gustaría aprender un poco más desde el punto de vista de la programación en Python. Considero que sin código, los modelos no pueden funcionar, pero el código no puede existir sin fundamentos matemáticos.

Para terminar, me gustaría hablar del gran debate que hay en la sociedad. Las personas tienen cierto temor a la Inteligencia Artificial. ¿Se perderán muchos puestos de trabajo? ¿Puede poner en peligro la intimidad de ciertas personas? Es claro que un mal uso de la Inteligencia Artificial producirá problemas en el mundo; pero ¿por qué nos limitamos a ver las desventajas? Pienso que la Inteligencia Artificial va a mejorar nuestras vidas en un futuro cercano, produciendo grandes avances en la ciencia. Simplemente hay que adaptarse al cambio.

Bibliografía

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[3] Francois Chollet. *Deep learning with Python.Second Edition.* Manning, 2021.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Wolfgang Karl Härdle and Léopold Simar. *Applied multivariate statistical analysis.* Springer Nature, 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[8] Amirhossein Kazemnejad. Transformer architecture: The positional encoding. *kazemnejad.com*, 2019.

[9] Andre Martins and Ramon Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International conference on machine learning*, pages 1614–1623. PMLR, 2016.

[10] John McCarthy. What is artificial intelligence. *Stanford*, 2007.

[11] Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[12] T Krishna Mohana, V Lalitha, L Kusuma, N Rahul, and M Mohan. Various distance metric methods for query based image retrieval. *Int J Eng Sci*, 5818, 2017.

[13] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021.

[14] Christopher Olah. Deep learning, nlp, and representations. https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/, 2014.

[15] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[16] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[17] Bernhard Schölkopf. The kernel trick for distances. *Advances in neural information processing systems*, 13, 2000.

[18] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[20] William F Trench. *Introduction to real analysis*. Trinity University, 2013.

[21] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.