Estudio de la capacidad de los sistemas Transformers en problemas de diferente complejidad algorítmica y modelado del lenguaje humano

Sonia Cambrón Blanco

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Directores del trabajo: Álvaro Lozano Rojo y Antonio Miguel Artiaga

12 de junio de 2024

Summary

Artificial intelligence is a technology that enables computers to simulate human intelligence and problem solving capabilities. A principal branch of artificial intelligence is Machine Learning. It focuses on the data and algorithms that enable artificial intelligence to imitate the way that humans learn, gradually improving its accuracy. In other way, Deep learning is a subset of machine learning that uses multi-layered neural networks to simulate the complex decision making power of the human brain.

In this dissertation, we study Transformers. A Transformer is a Deep Learning architecture that processes a sequence of discrete symbols or continuous values through the use of functions and finally transforms it into another sequence.

First, this architecture makes a representation of the initial sequence. The input sequence is encoded with symbols to sequence of integers called tokens. After, input tokens are mapped to a sequence of vectors, called Embeddings. This representation is the input of the following steps.

Then, Transformers use attention models to prioritize relevant information. Attention mechanisms enhance Transformers by focusing on important input elements, improving prediction accuracy. The model focuses on one component within the network's architecture that is responsible for creating interdependent relationships within input elements, this type of attention is called Self Attention. Self Attention consists of nonlinear functions that use dot products.

Finally, Transformers use a Feed Forward Network. Feed Forward Networks are neural networks that pass information forward through the network, from the input layer, through the hidden layers, and finally to the output layer. In a Transformer, the Feed Forward operation is a key component that takes the output of the Self Attention mechanism and transforms it into the final sequence of the model.

By combining tokenization, embedding, Self Attention, and Feed Forward networks, Transformers process and transform sequences effectively.

There are three types of Transformers: encoder, decoder and encoder decoder.

An encoder is a model that processes a sequence of tokens and generates a representation of the input sequence, which can be used for classification. The encoder is concerned with the input sequence and does not generate any output sequence.

A decoder takes in a representation of the input sequence and uses it to generate a sequence of tokens individually, with each token being conditioned on the previously generated tokens.

An Encoder Decoder has an encoder and a decoder. The encoder takes in the input sentence and produces a representation of it, which is then fed into the decoder to generate the output sentence. The decoder uses a attention mechanism that is applied to both the output of the encoder and the input of the decoder.

Transformers have revolutionized the machine learning and artificial intelligence by providing a powerful architecture for sequence processing. Due to their ability to capture complex relationships in input sequences, Transformers remain a subject of research and development.

Índice general

Capítulo 1

Introducción

La inteligencia artificial (IA) es una combinación de algoritmos que tienen como objetivo crear máquinas con las mismas capacidades que un ser humano. Para ello necesitan aprender a resolver problemas de manera automática a partir del procesado de la información.

Debido al gran interés en la IA, surgió el concepto de *Deep Learning*. *Deep Learning* [17][22] se refiere a un tipo de *Machine Learning* que se ajusta a los datos, utilizando una red neuronal de un gran número de capas y con más niveles de abstracción, y que permite resolver problemas de mayor dificultad. Algunos de los algoritmos más conocidos de *Deep Learning* permiten extraer información sobre imágenes, generar imágenes a partir de texto, procesar texto o voz en un asistente de voz.

Uno de los problemas más habituales en *Machine Learning* es el modelado predictivo [26], que busca crear algoritmos que a partir de unos datos introducidos busquen patrones, para posteriormente predecir el resultado con datos nuevos a partir de la hipótesis creada. Al proceso de búsqueda de patrones se le llama **aprendizaje** y a los datos introducidos les llamamos **datos de entrenamiento**. A los nuevos datos que queremos predecir les llamamos **datos de test**.

El modelado predictivo se puede asemejar a un problema de aproximación de funciones, una regresión o una interpolación, entre otras. Se busca una transformación $f : \mathbb{R}^n \to \mathbb{R}^m$ de manera que $f(x) \approx y$ para los datos de entrenamiento. La medida de proximidad entre la predicción $f(x)$ y el valor real y se denomina **función de pérdida** y podemos calcularla de varias formas dependiendo el problema que se esté tratando, como por ejemplo la norma de la diferencia.

La información explicada en este capítulo puede ser ampliada en [2], [25] y [27].

1.1. Introducción a la regresión lineal

Un método clásico de modelado predictivo es la regresión lineal [5] [12]. Este algoritmo permite predecir el comportamiento de una variable a partir de otra, asumiendo una relación lineal. Intuitivamente, se dice que existe regresión lineal de los valores de una variable con respecto a los de la otra cuando hay una línea llamada línea de regresión que se acerca mucho a los valores observados. Cuando esto ocurre, decimos que la línea de regresión se ajusta a los valores. La regresión se utiliza para identificar posibles relaciones o para predecir una variable a partir de otra, creando hipótesis sobre la relación de las variables.

El cálculo de la línea de regresión [1] [9] que mejor representa los puntos observados se puede hacer de diversas maneras. Se quiere buscar aquella cuya distancia sea mínima desde los nuevos puntos y por lo tanto, la que minimice el valor de la función de pérdida. Asumiendo una dependencia lineal, el método más común es medir la distancia vertical desde cada punto hasta la recta. El modelo de regresión lineal simple tiene la siguiente forma.

$$y_i = \alpha + \beta x_i + \varepsilon_i,$$

donde $X = (x_1, \ldots, x_n), Y = (y_1, \ldots, y_n)$ son los valores reales, α es la ordenada en el origen, β es la pendiente de la recta y ε es la variable de las perturbaciones, ya que normalmente no se va a establecer una relación lineal exacta entre ellas.

Para estimar el modelo, buscamos una recta

$$\hat{Y} = \alpha + \beta X,$$

que se ajuste a los puntos.

Luego, utilizaremos el método de mínimos cuadrados para crear esta recta, de forma que minimizaremos la suma de los cuadrados de los errores con la siguiente fórmula. Esta suma de los errores será la **función de coste**.

$$f(x,y,\theta) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \alpha - \beta x_i)^2,$$

donde $\theta = (\alpha, \beta)$.

Finalmente, obtenemos que $E(Y) = \alpha + \beta X$. A partir de la resolución de la fórmula utilizando mínimos cuadrados, tenemos que los valores α y β que minimizan el error son los siguientes:

$$\theta = (\alpha, \beta) = (X'^{\intercal} X')^{-1} X'^{\intercal} Y,$$

donde $X' = (1, X)$.

Ejemplo 1. Este método se puede aplicar a otro tipo de dependencias en las que podemos linealizar el modelo en sus parámetros mediante transformaciones. Por ejemplo, si tenemos una función exponencial, entonces utilizaremos la siguiente expresión para el modelo de regresión lineal

$$Y = \omega \cdot e^{k \cdot X + \varepsilon},$$

donde k y ω son los sesgos y los pesos y ε es la perturbación.

Ejemplo 2. La regresión logística [13] es un tipo de regresión que busca predecir una característica cualitativa utilizando datos de otras variables conocidas.

En la mayoría de los casos, se utiliza la regresión lineal, que es un método ampliamente utilizado para establecer una variable en relación con otra. Desde el punto de vista aritmético, la regresión lineal funciona correctamente. Sin embargo, en situaciones en las que la variable a explicar solo puede tener dos valores, es decir, la existencia o ausencia de un proceso específico, al evaluar la función para valores específicos de las variables independientes, se obtendrá un número que será diferente de 1 y de O, lo cual no tiene sentido en absoluto. La regresión lineal no funciona en este caso, mientras que la regresión logística se adapta adecuadamente a esta situación.

Como hemos visto en el anterior ejemplo, la regresión lineal no va a funcionar en todos los casos. Por ello, surgieron nuevos algoritmos que resolvían problemas de dependencia no lineal. El algoritmo con mayor importancia es el método de optimización del gradiente, ya que optimiza funciones no lineales. Primero, veamos qué es el gradiente.

Definición. Dado un $x, y \in \mathbb{R}^n$, $\theta = (\alpha, \beta)$ y una función $f : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^2 \to \mathbb{R}$ derivable, el gradiente de f con respecto a θ se define como el vector cuyas componentes son las derivadas parciales de f con respecto a θ:

$$\nabla f(x,y,\theta) = \left(\frac{\partial f}{\partial \alpha}(x,y,\theta), \frac{\partial f}{\partial \beta}(x,y,\theta) \right).$$

El gradiente de una función devuelve la dirección donde la función tiene un máximo crecimiento.

En este método, vamos a tener un vector de decisión $\theta = (\alpha, \beta)$, que a partir de un punto θ^0 inicial, vaya modificándose en un número finito de iteraciones para finalmente devolver una solución θ^*. Este número finito es fijado en el entrenamiento y en el capítulo final de resultados lo vamos a llamar **iteraciones**. Este proceso se realizará mediante la siguiente fórmula.

$$\theta^{k+1} = \theta^k - \gamma \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla f(x_i, y_i, \theta^k),$$

donde k es la iteración actual, γ es el tamaño del paso de búsqueda, $x, y \in \mathbb{R}^N$ y $\nabla f(x,y,\theta)$ denota el gradiente de la función $f(x,y,\theta)$ con respecto a θ.

1.2. Redes MLP

Para modelar un problema con *Machine Learning* vamos a seguir los siguientes pasos: escoger un conjunto de datos de entrenamiento completo, elegir el método de optimización y los parámetros iniciales, entrenar el modelo y optimizar el modelo mediante los datos de test. Normalmente, este proceso lo vamos a tener que repetir varias veces para conseguir probabilidades de acierto altas.

Dentro de los parámetros iniciales, tenemos que decidir cuantas capas queremos que tenga la red. Si elegimos 2 capas, la entrada x pasará primero por la red neuronal devolviendo una salida x' y después x' pasará de nuevo por la red neuronal devolviendo una salida final.

La redes MLP [6] son redes neuronales que tienen varias capas, la primera es una capa de entrada, la última una de salida y las demás son capas ocultas, y que entre ellas se conectan completamente para realizar el aprendizaje. Nos referimos a la capa de entrada como la capa que tiene como entrada la x, a la capa de salida como la que devuelve la solución final y a las capas ocultas como las capas que hay entre la capa de entrada y la de salida. Además, esta red utiliza el método del gradiente para la optimización de los resultados.

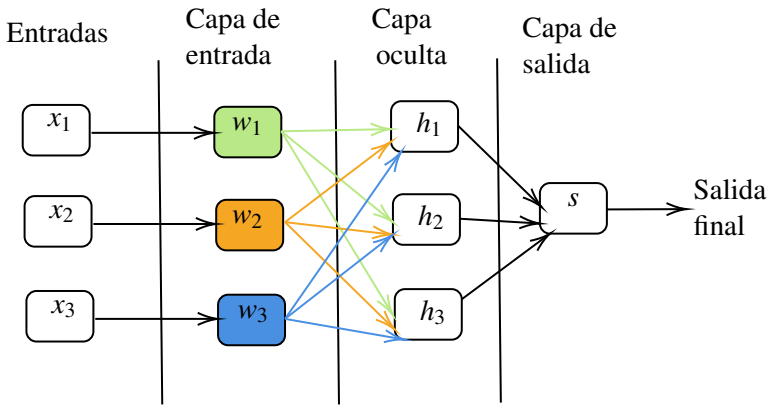


Figura 1.1: Capas en una red MLP.

Por ejemplo, la regresión logística (explicada en el ejemplo 2) se asemeja mucho a las redes MLP, sin embargo, no usa ninguna capa.

A partir de una entrada $x \in \mathbb{R}^n$ y una salida $y \in \mathbb{R}^m$, se busca una transformación f que desconocemos, de la forma

$$f : \mathbb{R}^n \to \mathbb{R}^m$$
$$f(x) \approx y.$$

A partir de muchas entradas (x, y), se define una función F de forma que exista $\varepsilon \in \mathbb{R}$ lo más pequeño posible tal que $|F(x) - f(x)|_\infty < \varepsilon$ para todo $x \in \mathbb{R}^n$. Después del entrenamiento, se predicen los resultados de los datos de test. A partir de las entradas x vamos a obtener un resultado $F(x)$, que compararemos con y. Habitualmente, se guarda si los resultados han sido correctos o no para, después de comparar todos los datos, calcular la **probabilidad de acierto**. Por lo tanto, es muy importante tener muchos datos de entrenamiento y lo más variados posibles, ya que van a tener un papel relevante en la creación de la función F. Además, a partir del siguiente resultado, podemos afirmar que va a existir una red neuronal que se aproxime a f.

Teorema 1.1. *(**Teorema de aproximación universal**): [10] Sea K un subconjunto compacto de $\mathbb{R}^n$, $\varepsilon > 0$ y cualquier función f continua en K, existe una red neuronal MLP con una capa oculta $F : \mathbb{R}^n \to \mathbb{R}$ tal que para todo $x \in K$ cumple que*

$$|F(x) - f(x)| < \varepsilon.$$

Actualmente, se usan un conjunto de técnicas matemáticas, llamadas mecanismos de atención, para ayudar a detectar las partes de la entrada que se consideran más importantes en la creación de las relaciones entre los elementos de la cadena y los patrones que siguen.

1.3. Mecanismos de atención

Los mecanismos de atención son técnicas que permiten a las redes neuronales dar prioridad a una parte de los datos de la entrada y disminuir la importancia de otras partes. Por ello, necesitamos dos matrices C y M que guarden las relaciones obtenidas hasta el momento, seleccionando las partes que son más relevantes para la solución del problema. Estas matrices se modifican tras cada nueva entrada y a partir de estos nuevos datos, crearán la salida.

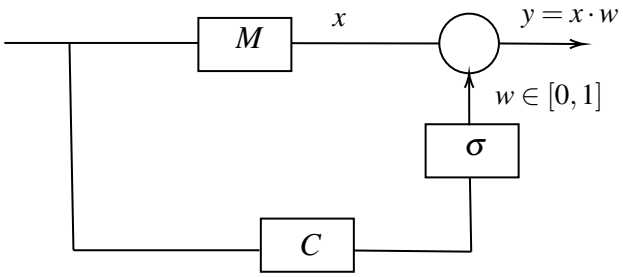


Figura 1.2: Modelo de atención con σ la función sigmoidea y C y M las matrices bloque que guardan la prioridad.

Este tipo de técnicas ya han sido introducidas en redes neuronales anteriores al Transformer. Una de esas redes fueron las *Highway Networks* [29]. Este modelo se caracteriza por el uso de funciones que aprenden a regular la información a través de una red con varias capas. Consta de L capas donde la l-ésima capa ($l \in \{1,2,\dots,L\}$) aplica tres transformaciones no lineales en su entrada x_l para producir y_l. $T(x_l, W_T)$ se denomina la transformación y $C(x_l, W_C)$ es el acarreo, ya que expresan qué parte de la cadena de salida es producida por transformar la entrada (T) y cuál por transportarla (C). Llamamos $H(x_l, W_H)$ a una transformada afín seguida de una función de activación no lineal, entonces

$$y_l = H(x_l, W_H) \cdot T(x_l, W_T) + x_l \cdot C(x_l, W_C).$$

A partir de la ecuación anterior, vamos a obtener una salida y_l que en la siguiente capa actuará como entrada, es decir, como x_{l+1}. Esta fórmula requiere que la dimensión de x_l, y_l, $H(x_l, W_H)$, $T(x_l, W_T)$ y $C(x_l, W_C)$ sea idéntica.

Con respecto a la estructura del trabajo, en el capítulo 2 veremos qué es un Transformer, las partes que lo forman y los tipos de Transformer que existen. En el capítulo 3 aplicaremos el modelo Transformer a algunos problemas matemáticos y analizaremos los resultados que hemos obtenido.

Capítulo 2

Transformers

Definición. Un **Transformer** es un modelo de *Deep Learning* que procesa una secuencia de símbolos discretos o valores continuos mediante la utilización de mecanismos de atención y finalmente la transforma en otra secuencia.

Este modelo se caracteriza por trabajar con entradas de diferente longitud, transformándolas en una serie de vectores que tienen la misma dimensión.

En este capítulo vamos explicar el funcionamiento de un sistema Transformer. Primero, vamos a entender qué operaciones y en qué orden las hace y posteriormente nos centraremos en detallarlas. El contenido de este capítulo se basa fundamentalmente en [2], [7], [25] y [28].

Primero, un Transformer crea una representación de la secuencia inicial. Los símbolos de la secuencia de entrada se codifican en una secuencia de enteros, llamados tokens. Después, los tokens de entrada se mapean a una secuencia de vectores, llamada Embedding. Esta representación es la entrada de los siguientes pasos.

Luego, los Transformers utilizan modelos de atención para priorizar la información relevante. Los mecanismos de atención mejoran los Transformers al enfocarse en elementos importantes de la entrada, mejorando la precisión de las predicciones. Este modelo de atención se centra en un componente dentro de la arquitectura de la red que es responsable de crear relaciones interdependientes dentro de los elementos de entrada, este tipo de atención se llama *Self Attention*. *Self Attention* consiste en funciones no lineales que utilizan productos escalares.

Finalmente, los Transformers usan una red neuronal *Feed Forward*. Las redes *Feed Forward* son redes neuronales que pasan información hacia adelante a través de la red, es decir, que la entrada de una capa es la salida de la anterior. En particular, vamos a utilizar las redes MLP que hemos explicado en el capítulo 1.2. En un Transformer, la operación de *Feed Forward* es un componente clave que toma la salida del mecanismo de *Self Attention* y la transforma en la secuencia final del modelo.

Al combinar la tokenización, *embedding*, *Self Attention* y la red *Feed Forward*, los Transformers procesan y transforman secuencias de manera efectiva.

Ejemplo 3. Supongamos que tenemos la siguiente cadena `'hola mundo'`. Además, tenemos un vocabulario $\{$`hola`, `mundo`$\}$. Primero, obtendríamos los dos tokens, que son `'hola'` y `'mundo'` y posteriormente crearíamos la representación $\{0,1\}$, que es el valor que le damos a cada token en el vocabulario. Es decir, le hemos dado a `'hola'` el valor 0 y a `'mundo'` el valor 1. Posteriormente, pasaríamos X= $\{0,1\}$ por el proceso de *embedding*, después por el método de atención y finalmente por una red neuronal MLP.

2.1. Tokenización

Lo primero que haremos será separar nuestra entrada en símbolos. Estos símbolos los llamamos **tokens** y nos van a servir para relacionar la cadena de entrada con nuestro vocabulario.

Para empezar, vamos a dividir nuestra entrada en caracteres y cada uno de ellos será un token que introduciremos en una lista. Posteriormente, calcularemos la relación entre los diferentes tokens, que

es la probabilidad de que dos tokens aparezcan juntos. Luego, elegiremos el de mayor probabilidad e introduciremos el nuevo token, que resulta de la concatenación de los dos tokens anteriores, en nuestra lista. Además, eliminaremos los tokens que ya no tengan ninguna aparición por separado en nuestra entrada.

Este proceso lo repetiremos hasta que los tokens correspondan a palabras de nuestro vocabulario. En caso de que no encuentre algún símbolo en el vocabulario y no consiga terminar, obtendremos un error ya que el vocabulario no estará completo para poder procesar esa secuencia. Veamos un ejemplo del proceso de tokenización.

Ejemplo 4. Sea la cadena de entrada $x =$ `'la laca'` y el vocabulario $\psi = \{\texttt{la}, \texttt{laca}\}$.

Primero, tendremos 3 símbolos: `'l'` con 2 apariciones, `'a'` con tres apariciones y `'c'` con 1 aparición. Ahora, vamos a calcular la probabilidad de que dos símbolos aparezcan juntos. Es decir, `'la'` tiene 2 apariciones, `'ac'` tiene 1 aparición y `'ca'` aparece una única vez. Por lo tanto, vamos a unir `'l'` con `'a'` y eliminaremos el símbolo `'l'` ya que no tiene ninguna aparición por separado.

Ahora, repetimos el proceso. Tenemos 2 veces la cadena `'la'`, 1 vez `'c'` y otra `'a'`. Como `'lac'` y `'ca'` tienen la misma probabilidad de ir juntas, elegimos una de las dos. Por ejemplo, unimos `'lac'` y eliminamos `'c'`. En este momento tenemos tres símbolos: `'la'`, `'lac'` y `'a'`. Realizando de nuevo este método obtendremos los dos tokens `'la'` y `'laca'`, que corresponden con palabras dentro del vocabulario y por lo tanto hemos terminado.

2.2. Embeddings

A continuación, vamos a crear una matriz que represente nuestra entrada a partir de los tokens creados en el paso anterior. Supongamos que tenemos un vocabulario ψ con una longitud $|\psi|$ y sea $N \in \mathbb{N}$ el número máximo de tokens que vamos a tener en la cadena de entrada. Como hemos dicho antes, podemos tener entradas de diferente longitud, por lo tanto, transformaremos todas las cadenas de entrada para que tengan longitud N, añadiendo una constante tantas veces como sea necesario, y siendo N mayor a todas las longitudes de los datos de entrenamiento. Para el entrenamiento, es necesario elegir longitud N fija ya que los datos se cogen en conjuntos de una longitud llamada **batch size** para favorecer el paralelismo. En caso de no fijar N, vamos a operar cada elemento individualmente y por ello se crearán menos relaciones y será muy lento computacionalmente. Sin embargo, cada dato de test se evalúa de forma individual, por lo tanto, para el proceso de test no es necesario fijar N.

Para empezar, tenemos una matriz $\mathrm{T} \in \mathbb{R}^{|\psi| \times N}$ donde la n-ésima columna corresponde a un vector que representa el token n. Este vector $\mathbb{R}^{|\psi| \times 1}$ tiene todas las entradas del vocabulario iguales a 0 menos aquella que corresponda con dicho token, donde habrá un 1. Además, sea $D \in \mathbb{N}$ el tamaño fijo que queremos asociarle a nuestra matriz de entrada, y Ω_e la matriz $\mathbb{R}^{D \times |\psi|}$ que contiene las relaciones aprendidas a partir de las entradas anteriores y que va modificándose constantemente mientras el Transformer siga aprendiendo.

Por lo tanto, la matriz resultante de este proceso es

$$X = \Omega_e \mathbf{T}.$$

2.3. Mecanismo de atención

2.3.1. Dot Product Self Attention

Tras el proceso de Embedding hemos obtenido una matriz $X = (x_1, \ldots, x_N) \in \mathbb{R}^{D \times N}$ con $x_i \in \mathbb{R}^{D \times 1}$ para $i = 1, \ldots, N$, y vamos a crear tres matrices. En primer lugar, creamos una matriz de valores $V \in \mathbb{R}^{D \times N}$, que se construye de la siguiente forma:

$$v_n = \beta_V + \Omega_V x_n \in \mathbb{R}^{D \times 1}, \quad n \in 1, \ldots, N,$$

con $\beta_V \in \mathbb{R}^{D\times 1}$ los sesgos y $\Omega_V \in \mathbb{R}^{D\times D}$ los pesos correspondientes. Durante el entrenamiento, tanto los sesgos β_V como los pesos Ω_V se modifican tras cada iteración para ajustarse a los valores de entrada (x,y).

También vamos a definir la matriz de consultas $Q \in \mathbb{R}^{D\times N}$ como podemos ver a continuación.

$$q_n = \beta_Q + \Omega_Q x_n \in \mathbb{R}^{D\times 1}, \quad n \in 1,\ldots,N,$$

con $\beta_Q \in \mathbb{R}^{D\times 1}$ los sesgos y $\Omega_Q \in \mathbb{R}^{D\times D}$ los pesos, que durante el entrenamiento, estarán también constantemente modificándose.

Por último, vamos a definir la matriz de claves $K \in \mathbb{R}^{D\times N}$ como,

$$k_n = \beta_K + \Omega_K x_n \in \mathbb{R}^{D\times 1}, \quad n \in 1,\ldots,N,$$

con $\beta_K \in \mathbb{R}^{D\times 1}$ los sesgos y $\Omega_K \in \mathbb{R}^{D\times D}$ los pesos asociados a X.

Una vez creadas la matriz de valores $V = (v_1,\ldots,v_N)$, y las matrices $Q = (q_1,\ldots,q_N)$ y $K = (k_1,\ldots,k_N)$ seguiremos el esquema que podemos ver en la figura 2.3.1. Primero, definimos la siguiente función $att : \mathbb{R}^{D\times N} \times \mathbb{R}^{D\times N} \longrightarrow (0,1)^{N\times N}$ como,

$$A = att(Q,K) = softmax_m(K^\intercal Q),$$

para cada fila $m \in \{1,\ldots,N\}$ de la matriz A y con la función softmax, que se define de la siguiente manera,

$$\text{softmax}_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}.$$

Esta función devuelve la matriz A cuya coordenada (m,n) es el peso que ejerce x_m sobre x_n, y por lo tanto, cumple que $\sum_{m=1}^{N} A_{m,n} = 1$ para todo $n \in \{1,\ldots,N\}$. El producto matricial de K y Q devuelve la similitud entre los vectores de estas dos matrices. Por lo tanto, los valores $A_{m,n}$ dependen de las similitudes entre el n-ésimo vector de las consultas Q y la matriz de claves K.

Finalmente, construimos la matriz resultante del proceso de atención $Sa = V \cdot A$. Esta matriz se puede crear también a partir del siguiente bloque de atención $sa_n : \mathbb{R}^{D\times N} \longrightarrow \mathbb{R}^{D\times 1}$, tal que

$$sa_n(x_1,\ldots,x_N) = \sum_{m=1}^{N} A_{m,n}\, v_m, \tag{2.1}$$

con $Sa = (sa_1(X),\ldots,sa_N(X))$.

A partir de esta fórmula, le damos un peso a cada vector de la matriz de valores V y los sumamos para calcular la salida. Este proceso lo realizamos para asignar prioridad a una parte de los datos de la entrada y disminuir la importancia de otras partes de la secuencia de entrada. A partir del cálculo de estas prioridades, estableceremos las relaciones entre los diferentes tokens.

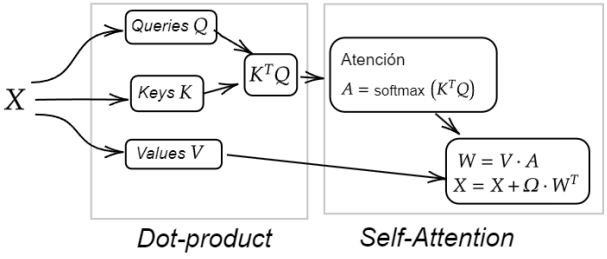


Figura 2.1: Esquema del proceso Dot Product Self Attention.

2.3.2. Multi-Head Self Attention

Un Transformer trabaja con grandes cantidades de información, por lo tanto, el uso de una única cabeza que realice la atención explicada en el apartado anterior puede resultar lento. Por ello, trabajaremos normalmente con varias **cabezas** que realizan el trabajo de forma simultánea en diferentes fragmentos de la entrada inicial. Este proceso se llama *Multi-head Self Attention*.

Sea $H \in \mathbb{N}$ el número de cabezas que tenemos y sea $h \in \{1,\dots,H\}$ la cabeza correspondiente, definimos las matrices de valores, consultas y claves de la siguiente forma:

$$V_h = \beta_h^V 1^T + \Omega_h^V X,$$
$$Q_h = \beta_h^Q 1^T + \Omega_h^Q X,$$
$$K_h = \beta_h^K 1^T + \Omega_h^K X.$$

Entonces, vamos a calcular el bloque de atención de cada cabeza por separado con sus respectivas matrices. Llamamos Ω_c a la matriz de combinación de las cabezas. Es decir, Ω_c almacena los valores de prioridad de cada cabeza y así, el resultado final dependerá de las salidas de todas las cabezas del mecanismo de atención en proporción a la prioridad que se le haya dado. Por lo tanto, definimos la matriz de atención como

$$MhSa[X] = \Omega_c[Sa_1(X),\dots,Sa_H(X)], \tag{2.2}$$

donde $Sa_h(X)$ es la matriz de la ecuación 2.1 en la capa $h \in \{1,\dots,H\}$.

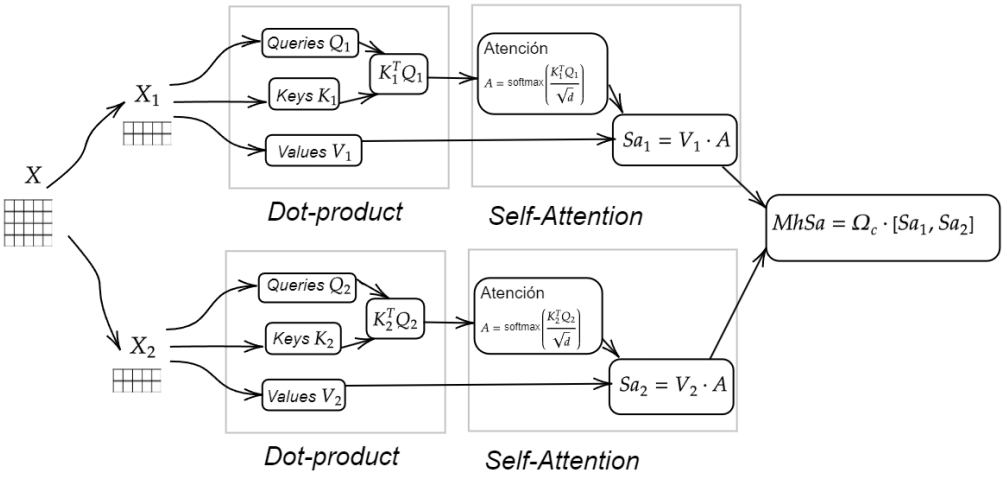


Figura 2.2: Estructura del proceso Multi-head Self Attention, donde d es el tamaño de las cabezas y $D =$<número de cabezas>$\cdot d$.

2.4. Red neuronal *Feed Forward*

Una **red neuronal *Feed Forward*** es un tipo de red neuronal artificial muy simple en la que la información se mueve hacia adelante desde la capa de entrada hasta la capa de salida, pasando por cada capa oculta. Podemos ver este tipo de red como una red MLP, explicada en la introducción en la sección 1.2.

Primero, definimos esta red neuronal cuya capa es

$$f(x) = \beta_2 + \Omega_2 \cdot ReLU(\beta_1 + \Omega_1 x),$$

para $x \in \mathbb{R}^{D\times 1}$, β_1 los sesgos y Ω_1 los pesos de la primera capa y β_2 y Ω_2 los de la segunda capa, donde

$$ReLU(z) = \begin{cases} 0 & \text{si } z < 0 \\ z & \text{en otro caso} \end{cases}.$$

En las redes MLP se utiliza la función sigmoidea en vez de la función ReLU.

En este proceso, obtendremos las relaciones entre los tokens de la secuencia, que en el proceso de test utilizaremos para producir nuevos tokens.

2.5. LayerNorm

Todo mecanismo de atención necesita normalizar los resultados obtenidos para que permanezcan en un dominio. Normalmente, se suele utilizar para normalizar los resultados tras el mecanismo de atención y para devolver una salida final. Hay varios tipo de normalización, por ejemplo, LayerNorm o BatchNorm. En un Transformer, vamos a utilizar LayerNorm [3] [24]. Veamos en que consiste dicha normalización.

Consideremos x el vector de representación de las entradas de una capa. Queremos obtener una salida que esté dentro del dominio [0,1], por lo tanto, fijando $\varepsilon > 0$ y los valores de la red neuronal β y Ω vamos a hacer la siguiente transformación lineal.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} \cdot \Omega + \beta,$$

donde μ es la esperanza de x y σ^2 es la varianza de x. Estos dos valores no son entrenados, sino que son valores constantes que se calculan para cada x y pueden ser calculados de la siguiente forma.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i,$$

$$\sigma^2 = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2},$$

donde N es la dimensión del modelo.

2.6. Procedimiento general

Las redes residuales [11] son redes neuronales que permiten la construcción de redes más complejas, haciendo uso de conexiones de salto que conectan capas no contiguas. Se trata de un procedimiento que comunica la salida con el valor de entrada inicial, sin necesidad de que éstos sean transportados por la red neuronal. Podemos ver el procedimiento en la siguiente figura.

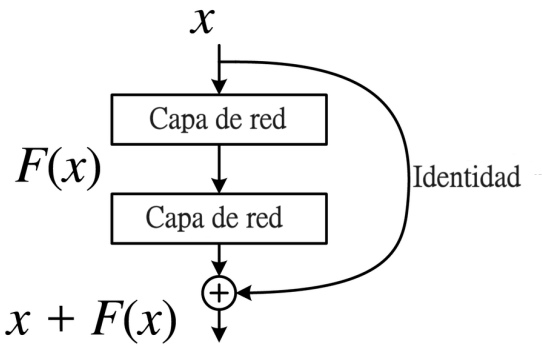


Figura 2.3: Estructura de una red residual.

Este método se demostró que funcionaba mejor para problemas más grandes [15] [23]. Esto se debe a que el gradiente de la función de pérdida puede retroceder directamente hacia la entrada a través de conexiones de salto. Por ello, se decidió utilizar este procedimiento en los modelos Transformer.

Finalmente, una vez que conocemos todos los pasos que el Transformer realiza para cada entrada, vamos a resumirlos.

Primero, se crean los tokens a partir de la cadena de entrada y se construye la matriz de *Embeddings*. Posteriormente, la siguiente función lineal $\mathbb{R}^{D\times N} \longrightarrow \mathbb{R}^{D\times N}$ aplica el mecanismo de atención a la matriz resultante del proceso de *Embedding*.

$$X \rightarrow X + MhSa(LayerNorm(X)),$$

con *MhSa* la ecuacion 2.2 definida anteriormente.

Finalmente, se utiliza una red *MLP* que introduce la X de la siguiente forma:

$$X \rightarrow X + MLP(LayerNorm(X)),$$

La salida final del modelo será la matriz X.

El pseudocódigo correspondiente a este proceso se puede ver a continuación.

Algorithm 1 Algoritmo de un Transformer

1: input: x el vector de entrada formado por los tokens, T el vector de relación entre los tokens y el vocabulario
2: $x = \Omega_e \cdot T$ // Proceso de embedding
3: $x = x + MhSa(LayerNorm(x))$ // Proceso de atención
4: $x = x + MLP(LayerNorm(x))$ // Red neuronal
5: **return** x

Este proceso define un bloque Transformer. Normalmente, un modelo Transformer tiene varios bloques Transformer. En modelos grandes, como por ejemplo los modelos de lenguaje, se va a llegar a tener centenares de estos bloques. Por ello, el Transformer se considera un modelo de Deep Learning.

Además del proceso del Transformer, hay otros factores que también influyen en el aprendizaje. Primero, es muy importante la elección de unos datos de entrenamiento variados y completos, de esta forma, el Transformer aprenderá de cada muestra en particular y obtendrá mejores relaciones. Por otro lado, los parámetros iniciales se tienen que ajustar a cada modelo. Por ejemplo, si queremos modelar un problema de texto, necesitaremos más capas de red neuronal, más cabezas de atención y mayor valor de D. Sin embargo, si queremos modelos más pequeños, como por ejemplo, un ejercicio de suma de dos números, con un Transformer pequeño nos bastará. Como podemos ver analizado en las siguientes referencias [4] y [32].

2.7. Tipos de modelos de Transformers

Existen tres tipos de Transformer, los encoder, los decoder y los encoder decoder. Tienen diferentes objetivos y por ello, tratan la información de entrada de diferente manera.

Las referencias usadas en esta sección son [2], [25] y [31].

2.7.1. Encoder

Este tipo de Transformer convierte los símbolos de la entrada en una representación numérica de forma que el mecanismo de atención es capaz de generar una salida a partir de esa representación. La representación de cada token depende tanto de los tokens anteriores como de los tokens posteriores.

Cuando se entrenan de forma aislada, la forma de entrenamiento más frecuente es la siguiente. A partir de una entrada (x, y), un encoder enmascara un número r de tokens de y. Posteriormente, se introduce en el modelo la entrada enmascarada. De esta forma, el modelo va a predecir los tokens enmascarados a partir de los tokens que se pueden ver y comparando el resultado obtenido con la cadena y inicial.

Para cada entrada, un encoder realiza la tokenización y el proceso de *embedding* para crear una representación numérica. Posteriormente, realiza el mecanismo de atención. Finalmente, pasa los resultados

por la red neuronal y a partir de la salida final predice qué token es el siguiente. Esta predicción corresponde al mayor valor en la salida final. Se puede aplicar sucesivamente este proceso y se le llamará **capas**.

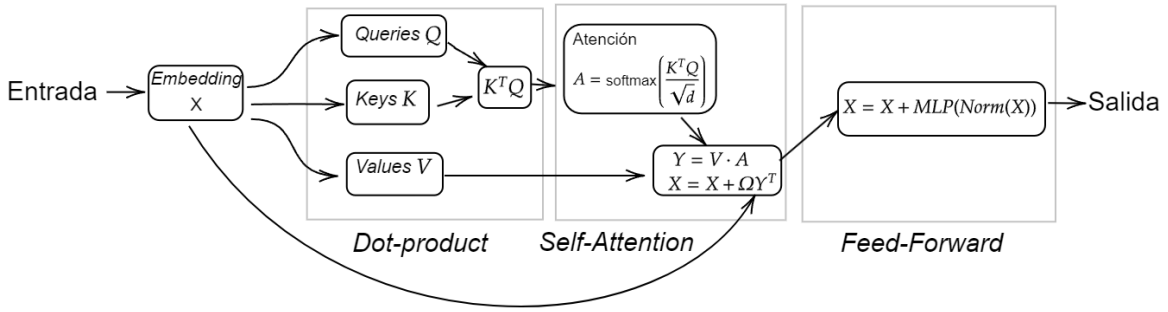


Figura 2.4: Estructura de un modelo encoder.

Ejemplo: BERT

El encoder BERT [19] [33] es un método de encoder basado en la combinación de un mecanismo de atención y una red *Feed Forward*. Este modelo consta de varios bloques de encoder apilados uno encima del otro. Cada bloque encoder consta de dos capas *Feed Forward* y una capa de *Self Attention*.

Cuando los datos pasan a través de estos bloques, se genera una matriz para cada secuencia de entrada, obteniendo información. Después de generar esa información, todos bloques del encoder se unen para obtener la salida. Cada bloque es responsable de establecer relaciones entre las representaciones de entrada y codificarlas en la salida.

Los modelos de encoder como BERT explotan el aprendizaje por transferencia. El objetivo en este aprendizaje es que el modelo aprenda información general. En la etapa de entrenamiento, la red resultante se ajusta para resolver una tarea en particular, utilizando un conjunto más pequeño de datos de entrenamiento.

BERT [25] utiliza un vocabulario de 30000 tokens. Los tokens de entrada son convertidos en embeddings de palabras de 1024 dimensiones ($N = 1024$) y pasados por 24 capas de red. Cada capa contiene un mecanismo de atención con 16 cabezas. Las consultas, claves y los valores para cada cabeza son de dimensión 64, es decir, $D = 64$.

2.7.2. Decoder

Un decoder predice el siguiente token a partir de una cadena de entrada. Este modelo coge como entrada los primeros $n-1$ tokens, predice el siguiente token y devuelve una salida con n tokens. Este proceso lo repetiremos hasta que tengamos una longitud máxima, que vendrá determinada por el número máximo de tokens que hayamos elegido inicialmente.

Un decoder es entrenado de la siguiente forma. Primero, recibiremos una entrada de $n+1$ tokens donde n tokens representan la entrada y el $n+1$ es el token que tiene que aprender a resolver. Cada entrada diferente va a ser ejecutada independientemente de forma que la red neuronal construya las relaciones de forma independiente y así poder minimizar el error.

Este modelo utiliza un tipo de atención llamado *Masked Self Attention*, que es un mecanismo de atención que calcula los tokens utilizando sólo datos del pasado. Como hemos visto, los tokens solo interaccionan con la red del Transformer durante la atención. Por ello, para asegurar que cada token solamente se fija en los anteriores, vamos a sumar una matriz $mask \in \mathbb{R}^{D\times N}$, que tiene todos los elementos superiores de la diagonal con $-\infty$, antes de hacer la ecuación 2.1. De esta forma las capas del Transformer solo pueden acceder a los tokens anteriores y al actual, sin verse afectado por los tokens que todavía no

conocemos. Además, el último token tiene los datos de todos los demás tokens, por lo que la salida final está relacionada con todos los tokens. Una matriz $mask \in \mathbb{R}^{3\times 3}$ sería de la siguiente forma:

$$mask = \begin{bmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{bmatrix}.$$

Para cada entrada, un decoder realiza primero la tokenización y el proceso de *embedding*. Después, realiza el mecanismo de atención *Masked Self Attention* en cada cabeza y finalmente genera la salida a partir de la red neuronal MLP. La estructura de un decoder la podemos ver en el siguiente diagrama.

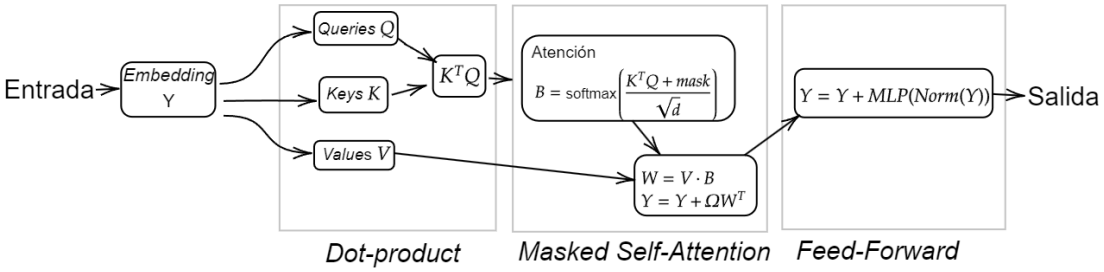


Figura 2.5: Estructura de un decoder.

Ejemplo: GPT3

GPT3 [18] [25] es un modelo de decoder que se centra en generar el siguiente token. GPT3 construye un modelo autorregresivo. Los modelos autorregresivos descomponen la distribución de una secuencia $(x_1,\dots,x_n)$ en un producto de distribuciones condicionales de la siguiente forma,

$$p(x_1,\dots,x_N) = \prod_{n=1}^{N} p(x_n|x_1,\dots,x_{n-1}).$$

A partir de esta fórmula, si fueran variables aleatorias discretas podríamos representar cada término de la fórmula en una tabla y estimar usando funciones simples de frecuencia. Sin embargo, a medida que va creciendo la secuencia, el tamaño crece exponencialmente. Por ello, vamos a simplificar la fórmula anterior suponiendo que cada distribución condicional es independiente a las observaciones previas excepto las L más recientes. Por lo tanto, ahora tenemos la siguiente fórmula,

$$p(x_1,\dots,x_N) = p(x_1)p(x_2|x_1)\dots p(x_L|x_1,\dots,x_{L-1}) \prod_{n=L}^{N} p(x_n|x_{n-L},\dots,x_{n-1}).$$

GPT3 calcula la probabilidad de que cada token sea el siguiente a añadir a partir de estas fórmulas. De esta forma, obtenemos una mejor probabilidad ya que está condicionada por los tokens anteriores.

En GPT3, la longitud de las secuencias es de 2048 tokens ($N = 2048$). Hay 96 capas de red neuronal, cada una de las cuales procesa un *embedding* de secuencias de tamaño 12288. Y hay 96 cabezas en los modelos de atención y la dimensión de las claves, los valores, las consultas son 128 ($D = 128$). Veamos a continuación un uso que le podemos dar a este modelo.

Ejemplo 5. Supongamos que cogemos la siguiente entrada '`GPT3 es`'. Entonces, el modelo GPT3 genera los siguientes tokens hasta que lleguemos a un total de 2048 tokens o cuando considere que ha llegado al final de la secuencia. Para cada token, el modelo generará las probabilidades de que cada palabra del vocabulario sea el siguiente token. Supongamos que tenemos un vocabulario $\psi = \{$`GPT3,`

`es, un, decoder, .`}. Entonces tras calcular las probabilidades obtendremos que ‘`un`’ tiene la mayor probabilidad y por lo tanto lo ponemos como el siguiente token.

Posteriormente, el siguiente token se generará a partir de la entrada ‘`GPT3 es un`’. De esta forma, llegaremos a generar la cadena completa ‘`GPT3 es un decoder.`’. Tras el previo entrenamiento del decoder, se ha creado una relación entre el final de la secuencia con ‘`.`’. Por ello, en este punto el decoder debería finalizar el proceso tras detectar ‘`.`’. En caso de que no se detectase el final, llegaríamos a un máximo de 2048 tokens, como hemos dicho anteriormente.

Este modelo puede realizar tareas como la generación de fragmentos de código basados en descripciones en lenguaje natural o aritmética. Sin embargo, cuando aumentamos el tamaño del modelo para resolver problemas mayores como por ejemplo la traducción entre idiomas, GPT3 no da buenos resultados.

2.7.3. Encoder Decoder

Por último, un Encoder Decoder es un tipo de Transformer que transforma una cadena de entrada en otra cadena de datos a partir de un método de atención. Requiere un encoder para crear una buena representación de la entrada y un decoder para generar el resultado correspondiente.

Primero, el encoder recibe la entrada y la procesa, generando una representación de cada token. Durante el entrenamiento, el decoder recibe la salida correcta, la introduce en la red mediante *Masked Self Attention* y predice la siguiente palabra en cada posición.

Posteriormente, vamos a crear una relación entre la salida del decoder y la salida del encoder. Para ello vamos a utilizar *Cross Attention*. *Cross Attention* es un tipo de atención que coge como entrada del vector de consultas (*Queries*) a la matriz final del decoder y para la entrada de valores (*Values*) y claves (*Keys*) la matriz final del encoder.

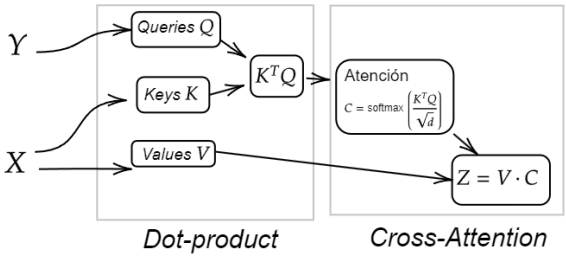


Figura 2.6: *Cross Attention* con X la matriz del encoder y Y la matriz del dedoder.

En un modelo encoder cada token interacciona con todos los demás tokens, luego tenemos una complejidad cuadrática. En cambio, en un modelo decoder cada token solo interacciona con los tokens anteriores, haciendo que haya un menor número de iteraciones. Sin embargo, el decoder a pesar de realizar menos iteraciones, también va a tener complejidad cuadrática. En grandes cadenas, con muchos tokens, la complejidad cuadrática puede afectar a nuestro ejercicio, llegando a resultar muy lento computacionalmente.

Como resumen, a partir de una entrada (x, y) se realiza de manera simultánea el encoder con entrada x y el decoder con entrada y. Finalmente, a partir del mecanismo de atención *Cross Attention*, vamos a unir los resultados para devolver la salida final. Podemos ver la estructura en la figura 2.7.

Ejemplo: Traducción

Las referencias usadas para este ejemplo son [4], [25] y [30].

La traducción entre idiomas requiere un encoder (para calcular una buena representación de la oración original) y un decoder (para generar la oración en el idioma de destino). Esta tarea se puede abordar utilizando un modelo encoder decoder.

Se considera la traducción del español al inglés. El encoder recibe la oración en español y la procesa a través de una serie de capas de red *Feed Forward* para crear una representación de salida para cada token. Durante el entrenamiento, el decoder recibe la traducción real en inglés y la pasa a través de las capas de la red *Feed Forward* que utilizan *Masked Self Attention*. Posteriormente, la capas de la red predicen la siguiente palabra en cada posición. Finalmente, el mecanismo de atención y la red *Feed Forward* crearán una unión entre la matriz del encoder y la matriz del decoder. En consecuencia, cada palabra de salida en inglés está condicionada a las palabras de salida anteriores y a la frase original en español.

Además, hay otros modelos para traducir textos como el que se muestra en la referencia [30]. En cuyo estudio se utiliza un modelo neuronal para el modelado de lenguaje. Este modelo tiene como objetivo estimar la calidad de la traducción de un dato de entrenamiento, sin necesidad de realizar comparaciones entre el texto inicial y el objetivo.

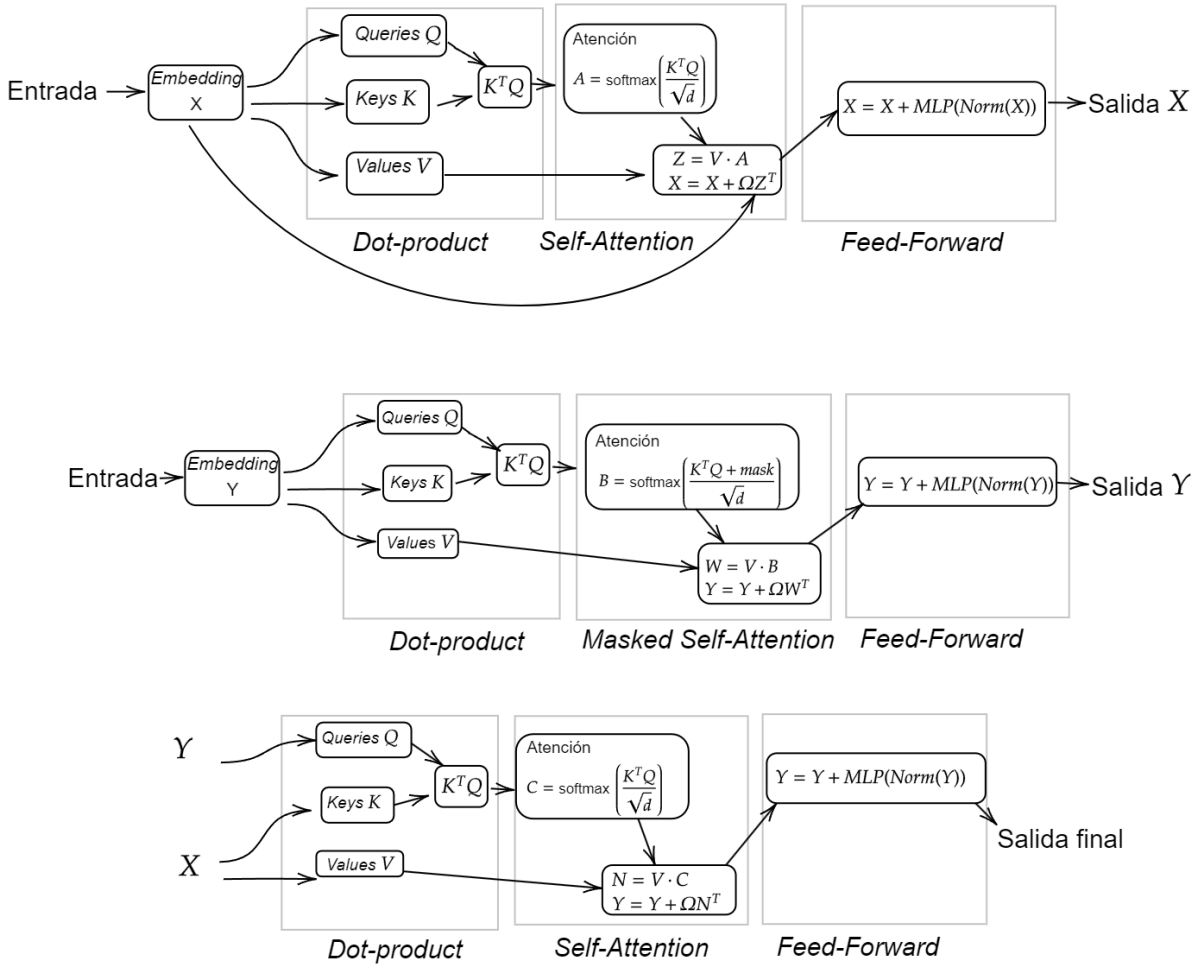


Figura 2.7: Estructura de un encoder decoder.

Capítulo 3

Resultados experimentales

En este capítulo, proporcionamos los resultados de la evaluación empírica. En el capítulo 2 hemos explicado los modelos Transformer. Además, hemos explicado los procesos de tokenización, *embedding*, *Self Attention* y red *Feed Forward*.

En las siguientes secciones, evaluamos las arquitecturas explicadas en la sección 2.7 con los siguientes tres experimentos y mostramos la precisión que hemos obtenido en cada uno. De esta forma, podemos evaluar si las arquitecturas pueden aprender el algoritmo correctamente.

3.1. Detección de paréntesis

En la mayoría de problemas matemáticos, utilizamos los paréntesis para separar cadenas o evaluar funciones. Naturalmente, las personas comprendemos mejor un problema si tiene más paréntesis, en cambio, para un Transformer, esto supone dos tokens más y no podemos deducir a simple vista que también le resulte más sencillo de entender. Por lo tanto, es importante saber si un Transformer es capaz de aprender qué elementos hay dentro de un paréntesis, para después aplicarle la función correspondiente.

En este ejemplo vamos a resolver el siguiente problema: supongamos que tenemos la cadena '2 (3 8 7) 4', entonces los elementos que están dentro del paréntesis son el 3, el 8 y el 7. Sea f la función que tiene que aprender y que es la siguiente:

$$y_i \equiv f(x_i) = \begin{cases} 1 & \text{si } x_i \text{ está dentro de los paréntesis} \\ 0 & \text{en caso contrario} \end{cases}.$$

Es decir, '2 (3 8 7) 4' lo resolveríamos como '0 0 1 1 1 0 0'.

Como dijimos en la introducción, primero vamos a crear una base de datos de entrenamiento, que en este ejercicio contendrá 10000 entradas. En ellas vamos a tener tuplas (x,y). Para la cadena anterior, tendríamos la tupla (x,y) con $x =$ '2 (3 8 7) 4 =' e $y =$'2 (3 8 7) 4 = 0 0 1 1 1 0 0'. Posteriormente, elegimos como método de normalización la función *LayerNorm* (explicada en la sección 2.5), como método de optimización el método del gradiente (véase más en la sección 1.1) y los parámetros iniciales. En este ejemplo, hemos escogido 1 cabeza, 1 capa, $N = 15$ como el número máximo de tokens y $D = 128$. Finalmente, entrenamos el decoder para después analizar los resultados obtenidos.

Tras el entrenamiento con una base de datos de test de 3000 muestras, hemos conseguido una probabilidad de acierto del 90,6 %. Por lo tanto, vamos a introducir en el decoder la cadena anterior $x =$'2 (3 8 7) 4 =', y predecir el resultado final para compararlo con $y =$'2 (3 8 7) 4 = 0 0 1 1 1 0 0'. Aunque la probabilidad de acierto es muy alta, no podemos garantizar que nos vaya a dar la solución correcta. La base de datos de test contenía 10000 muestras, luego, ha fallado en 940 muestras. Por lo tanto, no podemos afirmar que el decoder haya aprendido a resolver el problema en todos los casos.

Primero, hacemos el proceso de tokenización. Vamos a tener los siguientes tokens a partir de la cadena anterior: ['2', '(', '3', '8', '7', ')', '4', '=']. Posteriormente, creamos la representación numérica y realizamos la atención. De esta forma, se van a generar los siguientes 7 tokens, que corresponden a la

cadena de ceros y unos que contiene el resultado de la predicción. En la figura 3.1 se representa la matriz de atención obtenida para cada nuevo token que vamos a crear.

Podemos ver la matriz del método de atención cuando ya hemos predicho todos los elementos. Inicialmente, se ha introducido la entrada $x =$‘2 (3 8 7) 4 =’ y a partir del modelo decoder, ha generado los siguientes tokens. Hemos obtenido la siguiente matriz de atención tras haber generado la cadena completa, formada por ‘2 (3 8 7) 4 = 0 0 1 1 1 1 1’.

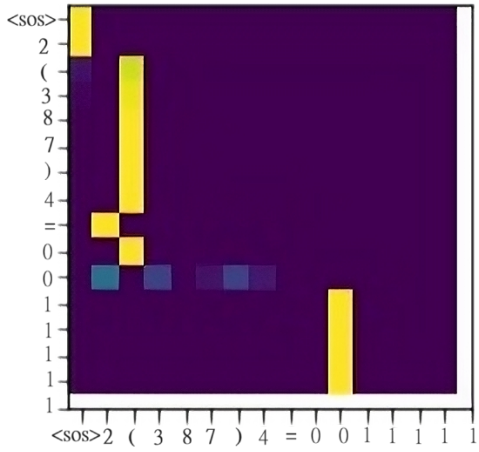


Figura 3.1: Matriz de atención con entrada ‘2 (3 8 7) 4 =’.

Analizando esta gráfica observamos los siguientes aspectos:

- El decoder concluye que ‘(3 8 7) 4’ se tiene que analizar todo junto y que los elementos que más importancia tienen son ‘3 8 7) 4’.
- Tras generar los tokens, no se obtiene un buen resultado ya que la cadena que ha predicho es ‘2 (3 8 7) 4 = 0 0 1 1 1 1 1’ e $y =$‘2 (3 8 7) 4 = 0 0 1 1 1 0 0’.
- Mientras se genera el siguiente token, el decoder detecta que ‘0 0 1 1 1 1’ se tiene que analizar todo junto.

Una vez analizada la atención, pasa por la red *Feed Forward* y obtendremos un vector en $\mathbb{R}^{11}$, que corresponde a la probabilidad que tiene cada token de ser elegido a continuación. El token que tenga mayor probabilidad es el que vamos a añadir a la cadena final y volveremos a repetir el proceso hasta que tengamos los 7 tokens que queremos.

3.2. Problema de máximos y mínimos

En esta sección queremos comprobar si un Transformer es capaz de resolver el problema de encontrar máximos y mínimos en un conjunto de números. Denotaremos por *‘MAX’* al máximo de dicho conjunto y por *‘MIN’* al mínimo.

Primero, vamos a entender el problema. Veamos el siguiente ejemplo $X =$‘MAX(3 5 MIN(9 2)) =’. Primero, evaluaríamos el mínimo de 9 y 2 y entonces tendríamos el máximo entre 3, 5 y 2. Y obtendríamos un 5.

Para el modelo, el espacio ‘ ’ delimita un nuevo valor en la cadena a introducir dentro de la función correspondiente y el paréntesis ‘)’ indica el fin de la cadena.

Para explicar el funcionamiento del decoder, vamos a utilizar únicamente el ejemplo X, con $N = 11$, $D = 1$ y un vocabulario $\psi = \{0,1,2,3,4,5,6,7,8,9,MAX(,MIN(,=,)\}$ con $|\psi| = 14$.

Una vez hemos entrenado el decoder con una base de datos de 3000 ejemplos, vamos a introducir nuestra entrada X y vamos a analizar los resultados obtenidos paso a paso.

Primero, convertimos la entrada X en tokens, obteniendo $X = [10,3,5,11,9,2,12,12,13]$. Veamos la representación de los elementos del vocabulario en la siguiente tabla.

Token	Código	Apariciones
'0'	0	0
'1'	1	0
'2'	2	1
'3'	3	1
'4'	4	0
'5'	5	1
'6'	6	0
'7'	7	0
'8'	8	0
'9'	9	0
'MAX('	10	1
'MIN('	11	1
')'	12	2
'='	13	1

$$\text{T=}\begin{pmatrix} \overbrace{}^{token\ 1} & & \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,1\,0\,0\,0 \\ 0 & 1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 1 & 0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,1\,0\,0\,0\,0\,0\,0 \\ 0 & 0\,0\,0\,0\,0\,1\,1\,0 \\ 0 & 0\,0\,0\,0\,0\,0\,0\,0\,1 \end{pmatrix}$$

Figura 3.2: Tokens de 'MAX(3 5 MIN(9 2))='.

Utilizando $D = 1$, tenemos la matriz $X \in \mathbb{R}^{1\times 11}$ inicial formada por los tokens y su representación numérica $X = \Omega \cdot T \in \mathbb{R}^{1\times 11}$ resultante del proceso de *embedding*.

$$X^{\intercal} = (-0{,}0773\ ,\ -0{,}4694\ ,\ 1{,}6875\ ,\ -0{,}8409\ ,\ 1{,}2996\ ,\ 0{,}3168\ ,\ 1{,}7468\ ,\ 1{,}7468\ ,\ -1{,}2071\ ,\ -0{,}2170)\,.$$

Podemos ver la representación de estas matrices en la siguiente figura.

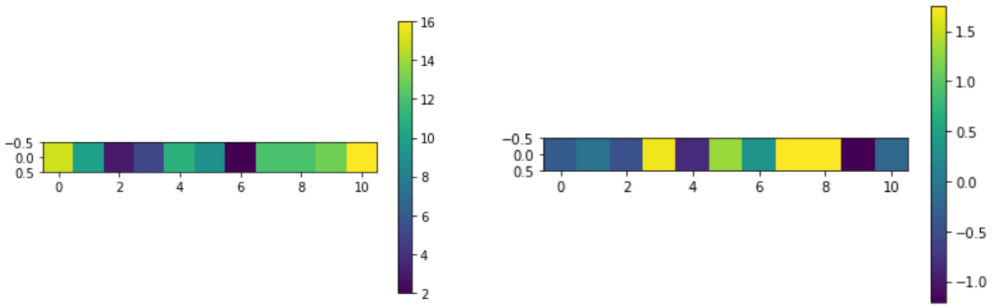


Figura 3.3: A la izquierda tenemos la matriz inicial X ='MAX(3 5 MIN(9 2)) =' y a la derecha la matriz X tras Embedding.

Una vez tenemos la nueva matriz X, vamos a realizar el mecanismo de atención. Como tenemos $N = 11$ y $D = 1$, los sesgos β estarán en $\mathbb{R}^{1\times 1}$, los pesos Ω estarán en $\mathbb{R}^{1\times 1}$ y las matrices de consultas (Q) , valores (V) y claves (K) estarán en $\mathbb{R}^{1\times 11}$. Veamos cuales son estos valores.

$$\begin{aligned} &\beta_Q = (0{,}0181) \quad \Omega_Q = (-0{,}4632) \quad Q^{\intercal} = 0{,}0435\cdot(1,1,1,1,1,1,1,1,1,1,1)\,, \\ &\beta_K = (-0{,}3551) \quad \Omega_K = (-0{,}2217) \quad K^{\intercal} = -0{,}3429\cdot(1,1,1,1,1,1,1,1,1,1,1)\,, \\ &\beta_V = (0{,}6578) \quad \Omega_V = (1{,}0065) \quad V^{\intercal} = 0{,}6025\cdot(1,1,1,1,1,1,1,1,1,1,1)\,. \end{aligned}$$

Durante el proceso de atención, vamos a generar la matriz $A = K^{\intercal}\cdot Q \in \mathbb{R}^{11\times 11}$. Finalmente realizaremos la operación softmax y obtendremos el vector final que será $B = V\cdot \text{softmax}(A) \in \mathbb{R}^{1\times 11}$.

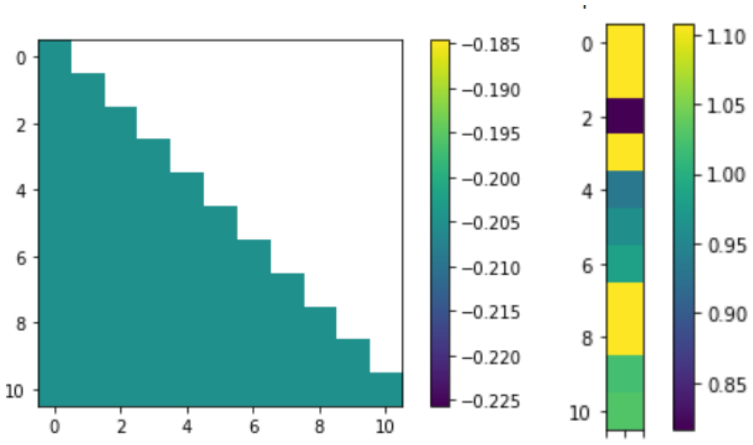
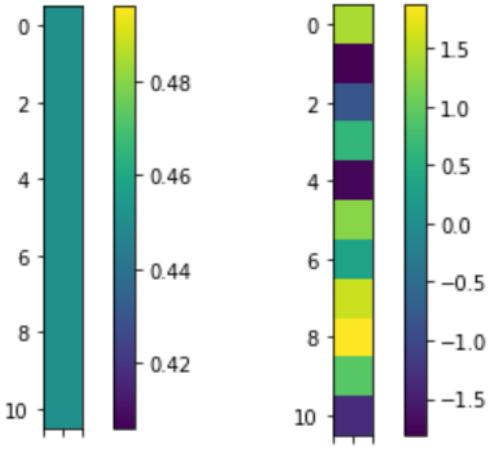


Figura 3.4: A la izquierda tenemos la matriz $A = K^{\intercal} \cdot Q$ y a la derecha el vector $B = \text{softmax}(A) \cdot V$.

El decoder utiliza *Masked Self Attention*, es decir, añade el valor -∞ en las posiciones correspondientes a los tokens posteriores. Por lo tanto, podemos ver como en la primera matriz de la figura 3.4 la matriz triangular superior está pintada en blanco, que representa el -∞. De esta forma, cada token solo interactúa con los tokens anteriores.

Una vez realizado el mecanismo de atención vamos a introducir $X = X + B$ en la red neuronal *Feed Forward*. Finalmente, el vector X final se puede ver en la figura 3.5. Este vector está formado por las probabilidades que tiene cada token de ser elegido. En este caso, vemos que el token que mayor valor tiene es el token 8, que corresponde con ')' en la cadena X. Para concluir, el decoder resolvería nuestro problema como *MAX(3 5 MIN(9 2)) =)*.



$$X = \begin{pmatrix} 1{,}3985 \\ -1{,}8081 \\ -0{,}8234 \\ 0{,}6279 \\ -1{,}7602 \\ 1{,}2105 \\ 0{,}3235 \\ 1{,}5745 \\ 1{,}8713 \\ 0{,}9107 \\ -1{,}3559 \end{pmatrix}.$$

Figura 3.5: A la izquierda el vector Z de Feed Forward y a la derecha el vector X final $(X = X + Z)$.

Como esperábamos, este resultado es erróneo ya que esta red está entrenada con un decoder muy pequeño y por tanto, no aprende este problema. A continuación, vamos a entrenar nuestra red neuronal para que aprenda la estructura de este programa.

3.2.1. Análisis de resultados y conclusiones

Construimos una base de datos de entrenamiento de 3000 muestras y una base de datos de test de 1000 muestras, con las operaciones (*MIN* y *MAX*). Vamos a estudiar el porcentaje de acierto tras entrenar el decoder con la base de datos. Primero, vamos a utilizar una máquina pequeña, con solamente 1 cabeza y 1 capa y después vamos a ir aumentando el decoder para mejorar los resultados. En la siguiente tabla hemos utilizado esta base de datos de entrenamiento para entrenar el decoder y posteriormente a partir de la base de test analizaremos las probabilidades de que el decoder devuelva el resultado correcto.

D	Batch	Capas	Cabezas	Tasa acierto
16	32	1	1	15.6 %
32	64	1	1	14.4 %
16	32	4	1	37.7 %
16	32	1	2	19.9 %
16	32	4	2	69.7 %
16	64	6	3	93.3 %
16	64	8	4	91.9 %
16	64	12	4	90.7 %

Cuadro 3.1: Tabla de resultados del problema de máximos y mínimos con una base de entrenamiento de 3000 muestras.

Como podemos ver en la tabla 3.1, vamos a conseguir que un decoder aprenda este problema. Hemos llegado a más de un 90 % de acierto con un decoder de 6 capas y 3 cabezas, es decir, no necesitamos un decoder muy grande para poder resolver este problema. Además, con este decoder de 6 capas y 3 cabezas, es necesario únicamente 100 iteraciones de la red neuronal para conseguir más de un 80 % de acierto, luego es un problema rápido y fácil de entrenar. Para menos cabezas y capas, no obtenemos buenos resultados, por lo tanto, podríamos afirmar que el mínimo decoder a utilizar sería de 6 capas y 3 cabezas.

Además, este problema necesita que el decoder aprenda qué representan los paréntesis y qué elementos hay dentro de los paréntesis. Por lo tanto, un decoder muy pequeño va a ser imposible que aprenda esta estructura debido a su complejidad. También, en este problema vamos a resolver primero una operación y después haremos el resto de operaciones utilizando este resultado. Esta composición de funciones *'MAX'* y *'MIN'* hace que el problema no sea lineal para una máquina y dificulta mucho la estructura y la creación de relaciones entre los tokens.

Por otro lado, si añadimos otras operaciones como puede ser la media o la suma modular, este problema se complica ya que estas operaciones no son tan intuitivas para un decoder y por lo tanto no tenemos buenos resultados. Ejecutando 1000 iteraciones de un decoder de 12 capas y 4 cabezas obtenemos únicamente un 70 % de acierto y por lo tanto no podríamos afirmar que el decoder ha entendido la estructura de este nuevo problema.

3.3. Simplificación de elementos inversos

Sea un grupo libre formado por a, b, c, d= a^{-1}, e= b^{-1}, f= c^{-1} y 1, vamos a simplificar una cadena de 6 elementos del grupo multiplicados entre sí. Es decir, si tenemos $a \cdot d = a \cdot a^{-1} = 1$ y finalmente hemos simplificado todo lo posible la operación dada. De la misma forma, $b \cdot d \cdot a \cdot e \cdot c = b \cdot a^{-1} \cdot a \cdot b^{-1} \cdot c = b \cdot b^{-1} \cdot c = c$ y hemos llegado a la mayor simplificación, pero en más pasos que antes.

Vamos a utilizar un vocabulario $\psi = \{1,a,b,c,d,e,f,=,<\text{pad}>,<\text{sos}>,<\text{eos}>\}$ con una longitud $|\psi|=11$. Como hemos vimos en la introducción, los Transformers se caracterizan por poder entrenar con datos de entrada de diferente longitud. Esto lo hacen mediante la constante de padding '$<$ pad $>$'. Este token se añade al final de cada entrada tantas veces como sea necesario para tener longitud N. Además, tenemos '$<$ sos $>$' (*start of sequence*), que representa el principio de la entrada y '$<$ eos $>$' (*end of sequence*), que representa el final de la entrada. Estos tokens son necesarios para que el decoder sepa la longitud de la cadena y calcule los tokens posteriores sin contar con los tokens de padding '$<$ pad $>$'.

Veámoslo con el siguiente ejemplo: X='a b e d b c =', donde $N = 15$ ya que tenemos 7 caracteres, 1 inicio de entrada, 1 final de entrada y 6 padding. Para empezar, vamos a inicializar $D = 16$, 1 capa de la red y 1 cabeza.

A continuación, creamos los tokens y nos queda [9, 1, 2, 5, 4, 2, 3, 7, 10, 8, 8, 8, 8, 8, 8], como podemos ver en la siguiente tabla.

Token	Código	Apariciones
‘a’	1	1
‘b’	2	2
‘c’	3	1
‘d’	4	1
‘e’	5	1
‘=’	7	1
‘< pad >’	8	6
‘< sos >’	9	1
‘< eos >’	10	1

Cuadro 3.2: Tabla de tokens de X=‘a b e d b c =’.

Una vez obtenidos los tokens, vamos a realizar la matriz de entrada mediante el proceso de *Embedding*. En la matriz de la figura 3.6 podemos ver cuales son las constantes de padding, ya que no modifican su valor en las filas correspondientes. Esto se debe a que el decoder ha encontrado el final de la cadena y por lo tanto no se le da importancia a estas constantes de padding.

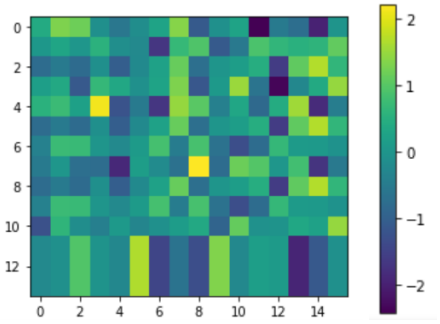


Figura 3.6: Valor de X tras el proceso *Embedding*.

Una vez obtenida X vamos a realizar el mecanismo de atención. Para ello vamos a obtener la matriz de consultas, valores y claves. Los sesgos de la matriz de consultas (Q), la matriz de claves (K) y la matriz de valores (V) son $\beta_Q = -0{,}3681$, $\beta_K = 0{,}2832$ y $\beta_V = -0{,}1890$ y las matrices de pesos correspondientes son las siguientes.

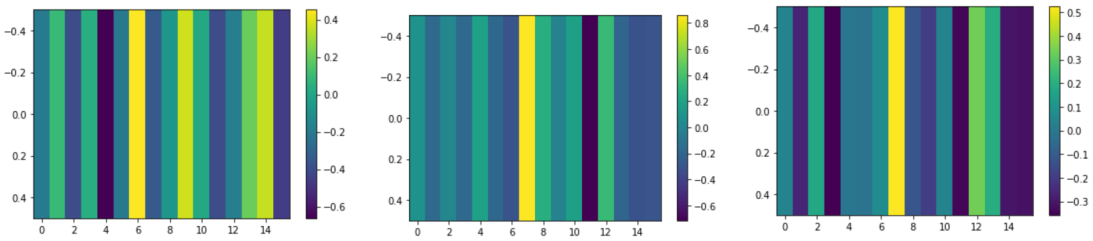


Figura 3.7: Pesos de la matriz Queries, Keys y Values.

Por último, realizando el mecanismo de atención vamos a obtener una matriz con $-\infty$ en la triangular superior.

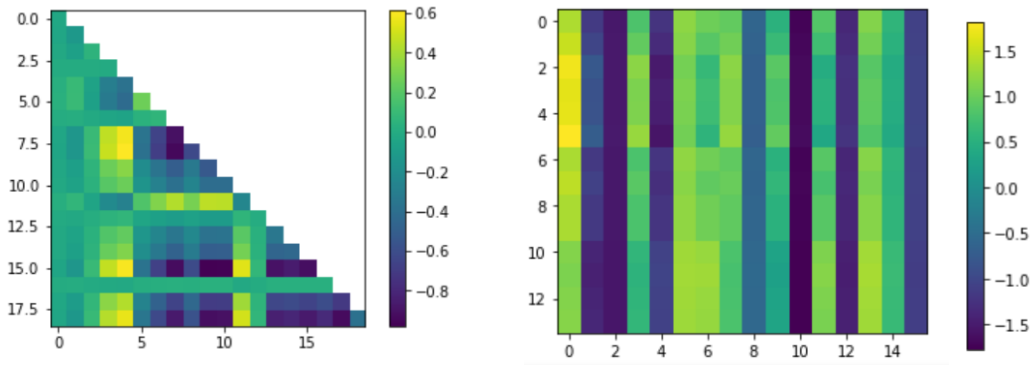


Figura 3.8: Salida de $K^{\intercal}Q$ a la izquierda y la matriz resultante de la atención a la derecha.

Una vez que hemos obtenido la matriz de atención, vamos a introducir nuestra matriz en la red *Feed Forward*.

Finalmente, obtenemos la matriz $X \in \mathbb{R}^{14\times 11}$ resultante de la realización del mecanismo de atención y el proceso *Feed Forward*, es decir, la matriz resultante del Decoder.

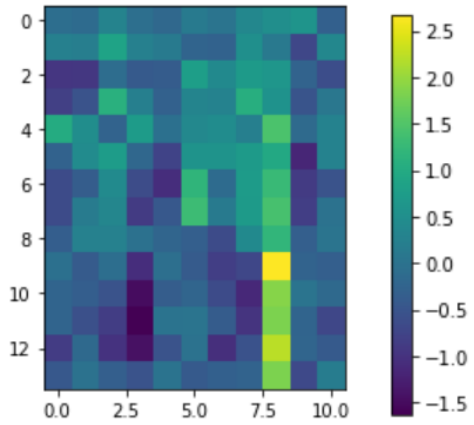


Figura 3.9: Matriz devuelta por el decoder para X=‘a b e d b c =’.

Analizando la matriz anterior, vemos que la columna con resultados más altos es la correspondiente al token 8. Es decir, el decoder ha devuelto que el último token es ‘=’ y por lo tanto la cadena final es ‘a b e d b c = a b e = d =’. Esto se debe a que con una capa y una cabeza no es capaz de aprender un problema de esta dificultad. Por ello, en la siguiente sección vamos a analizar el aprendizaje con diferentes tipos de parámetros iniciales, con la función de optimización del método del gradiente y con el método de normalización *LayerNorm* (para más información véase la sección 2.5).

3.3.1. Análisis de resultados

Creando una base de datos de entrenamiento de 2000 muestras y una base de datos de test de 300 muestras, hemos realizado un análisis de resultados en el que podemos ver el porcentaje de acierto en los test que hemos conseguido tras 10000 iteraciones, dependiendo el número de recursos que tenemos. En la siguiente tabla hemos utilizado esta base de datos de entrenamiento para el decoder y posteriormente a partir de la base de test analizar las probabilidades de que el decoder devuelva el resultado correcto.

D	Batch	Capas	Cabezas	Iteraciones	Tasa acierto
16	32	1	1	10000	6 %
32	64	1	1	10000	6 %
16	32	4	1	10000	44 %
32	32	4	1	10000	41 %
16	32	1	2	10000	8 %
16	32	4	2	10000	66 %
32	64	4	2	10000	63 %
16	64	12	4	100	85 %

Cuadro 3.3: Tabla de resultados con una base de entrenamiento de 2000 muestras.

Como podemos ver, con pocos recursos como son 1 cabeza y 1 capa, este ejercicio le resulta muy difícil al decoder ya que no tiene espacio para almacenar los resultados y por ello no realiza las relaciones de forma correcta. Tras un gran entrenamiento, solamente hemos conseguido un 6 % y no crece a medida que lo vamos entrenando, ya que realizando 5000 iteraciones, hemos obtenido el mismo resultado.

A medida que vamos aumentando las prestaciones, la probabilidad de acierto aumenta considerablemente utilizando las mismas iteraciones para entrenar el decoder. Aunque, como podemos ver en la tabla, no vamos a llegar a un gran resultado ya que como máximo hemos conseguido un 85 %. Además, se observa como en el último caso, donde hemos utilizado un decoder muy grande, tampoco es capaz de resolver el problema correctamente. Por ello, vamos a plantear otros métodos para mejorar este resultado.

3.3.2. Otros métodos

Podemos utilizar otra función de optimización llamada **adamW**. Tras varios estudios, como por ejemplo el estudio realizado en la referencia [8], se ha visto que resulta muy útil especialmente en problemas lógicos, como es nuestro caso. Este mecanismo añade una regularización a la función de coste que penaliza la norma 2 del vector de parámetros. De esta forma, los pesos se actualizan usando los gradientes de las funciones de pérdida en cada iteración. Véase más información sobre el algoritmo en el anexo A.

Como hemos dicho anteriormente, este método debería dar mejores resultados que el anterior, pero como vemos en la tabla, mediante este método no vamos a conseguir que el Transformer aprenda este ejercicio y por lo tanto no soluciona nuestro problema. En cambio, cuando tenemos un decoder grande como podría ser el último caso, llegamos a un porcentaje del 90 % de acierto. Por lo tanto, podríamos concluir que para nuestra base de datos este método nos proporciona buenos resultados para un decoder de más prestaciones. Sin embargo, no se podría concluir que este método es más útil que el anterior, ya que depende en gran medida de los parámetros iniciales.

Podemos ver a continuación las pruebas realizadas de este método.

D	Batch	Capas	Cabezas	Iteraciones	Tasa acierto
16	32	1	1	10000	4 %
16	32	4	1	10000	38 %
16	32	1	2	10000	19 %
16	32	4	2	10000	51 %
16	64	12	4	100	90 %

Cuadro 3.4: Tabla de resultados obtenida con optimización adamW, tras el entrenamiento con 2000 muestras y con 300 muestras de test.

Actualmente, en *Machine Learning* se trabaja con LLM (*Large Language Models*). Estos métodos nos ayudan a resolver problemas de gran complejidad en modelos Transformer. Las siguientes soluciones utilizan este tipo de métodos, que descomponen un problema grande en otros de menor tamaño.

Como hemos observado, una máquina pequeña no puede resolver este problema, por lo tanto vamos a utilizar estos métodos que se basan en dividir el problema en ejercicios más pequeños. Es decir, 'b a d e = 1' lo introduciríamos de la siguiente forma 'b a d e = b e = 1'. Realizándolo paso a paso, la máquina debería aprender mejor ya que realiza mejores conexiones al descomponerlo en cadenas más sencillas. En cambio, podemos ver otra vez que con un máquina muy pequeña, el modelo aprende peor que por el método inicial porque tiene una cadena más larga con más tokens. Sin embargo, a medida que vamos utilizando una mejor máquina los resultados aumentan considerablemente.

Veamos los resultados obtenidos mediante este método.

D	Batch	Capas	Cabezas	Iteraciones	Tasa acierto
16	32	1	1	5000	2.60 %
16	32	4	1	5000	40.9 %
16	32	1	2	5000	12.7 %
16	32	4	2	5000	58.4 %
16	64	12	4	100	91.3 %

Cuadro 3.5: Tabla de resultados obtenida a partir de los métodos LLM, tras el entrenamiento con 2000 muestras y con 300 muestras de test.

Por último, otra opción parecida a la anterior pero que podría funcionar mejor sería añadir paréntesis a los elementos que vamos a simplificar. Es decir, 'b a d e = 1' sería de la siguiente forma 'b a d e = b (a d) e = b e = (b e) = 1'. Esta solución tiene un vocabulario mayor, más tokens y una cadena más larga, lo cual es muy costoso y por lo tanto, no se consiguen buenos resultados para máquinas pequeñas. En cambio, esta solución es la más útil para nuestro problema, ya que podemos ver como en el último resultado hemos conseguido más de un 97 % de acierto.

Podemos ver los resultados obtenidos a partir de este método en la siguiente tabla.

D	Batch	Capas	Cabezas	Iteraciones	Tasa acierto
16	32	1	1	1000	2.6 %
16	32	4	1	1000	9.1 %
16	32	1	2	1000	1.6 %
16	32	4	2	1000	46.1 %
16	64	12	4	100	97.2 %

Cuadro 3.6: Tabla de resultados añadiendo paréntesis a las muestras, tras el entrenamiento con 2000 muestras y con 300 muestras de test.

3.3.3. Utilización de un Encoder Decoder

Un encoder decoder (véase más información en la sección 2.7.3) es un Transformer compuesto por un decoder y un encoder. Inicialmente, este problema lo hemos resuelto utilizando un decoder. Tras descubrir que no obtenemos buenos resultados, vamos a utilizar un encoder decoder. De esta forma, en el decoder seguimos prediciendo el siguiente token de uno en uno y en el encoder vamos a analizar la cadena completa.

Podemos ver los resultados obtenidos a partir de la siguiente tabla. Hemos utilizado la misma base de datos de 2000 muestras para el entrenamiento y 300 para el test.

D	Batch	Capas	Cabezas	Iteraciones	Tasa acierto
16	32	1	1	1000	16 %
16	32	4	1	1000	56 %
16	32	1	2	1000	29 %
16	32	4	2	1000	79 %
16	32	6	3	1000	98 %
16	32	12	4	100	96 %

Cuadro 3.7: Tabla de resultados obtenidos a partir de un encoder decoder, tras el entrenamiento con 2000 muestras y con 300 muestras de test.

Como podemos observar, utilizando menos iteraciones hemos llegado a un porcentaje de acierto mucho mayor que con el decoder inicial. Además, para un encoder decoder de 6 capas y 3 cabezas los resultados son de casi un 100 % de acierto, por lo tanto, podemos garantizar que con esta base de datos, se ha aprendido la estructura del problema.

3.4. Conclusión

Una vez realizados estos experimentos, vamos a analizar los resultados obtenidos.

En el primer experimento, hemos tratado un problema muy sencillo que consiste en detectar en una secuencia qué elementos hay dentro de un paréntesis. Este experimento lo hemos probado para un decoder muy pequeño de únicamente una capa y una cabeza. A partir de este decoder hemos obtenido porcentajes de acierto muy alto y hemos podido ver relaciones entre los paréntesis y los unos de la salida.

En el segundo experimento, hemos tratado un problema de máximos y mínimos que no funciona tan bien para Transformers muy pequeños. En cambio, con un decoder de 3 cabezas y 6 capas hemos conseguido un porcentaje de acierto muy alto.

Finalmente, para el tercer experimento, concluimos que simplificar una cadena de elementos de un grupo es un problema muy complicado para un Transformer pequeño. Tanto el decoder como el encoder decoder nos han proporcionado porcentajes de acierto muy bajos para máquinas pequeñas. En cambio, a medida que aumentamos tanto las capas como las cabezas, mediante el encoder decoder vamos a obtener mejores resultados. Además, utilizando algoritmos LLM hemos observado resultados muy similares a los obtenidos con el encoder decoder.

En conclusión, para cada experimento hemos encontrado un Transformer que devuelve buenos resultados y podríamos decir que ha aprendido la estructura del algoritmo.

3.5. Agradecimientos

Este trabajo ha sido parcialmente financiado por el Programa de Becas y Ayudas del Instituto de Investigación en Ingeniería de Aragón (I3A) y por la Cátedra BTS Group - Nuevas Tecnologías de Telecomunicaciones

Apéndice A

Pseudocódigo de AdamW

La referencia usadas en este anexo es [21].
Los parámetros definidos por defecto son los siguientes:

$$\gamma = 0{,}001,\ \beta = (0{,}9, 0{,}999),\ \varepsilon = 1e-08,\ \lambda = 0{,}01,\ amsgrad = False\ y\ maximize = False.$$

El algoritmo que sigue esta función es lo podemos ver a continuación.

input: γ(lr), β_1, β_2 (betas), θ_0 (parámetros), $f(\theta)$ (función objetivo), ε (epsilon), λ (caída de peso), *amsgrad*, *maximize*
initialize: $m_0 \leftarrow 0$ (primer momento), $v_0 \leftarrow 0$ (segundo momento), $\hat{v_0}^{max} \leftarrow 0$
for t=1 to ... **do**
    if *maximize* **then**
        $g_i \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
    else
        $g_i \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    end if
    $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\,\theta t - 1$
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
    $\hat{m}_t \leftarrow m_t / (1-\beta_1^{\mathsf{T}})$
    $\hat{v}_t \leftarrow v_t / (1-\beta_2^{\mathsf{T}})$
    if *amsgrad* **then**
        $\hat{v_t}^{max} \leftarrow \max(\hat{v_{t-1}}^{max}, \hat{v}_t)$
        $\theta_t \leftarrow \theta_t - \gamma\hat{m}_t / \left(\sqrt{\hat{v_t}^{max}} + \varepsilon\right)$
    else
        $\theta_t \leftarrow \theta_t - \gamma\hat{m}_t / \left(\sqrt{\hat{v}_t} + \varepsilon\right)$
    end if
end for
return θ_t

Bibliografía

[1] T. ALCALÁ, *Apuntes Regresión lineal - Universidad de Zaragoza*.

[2] C. M. BISHOP, H. BISHOP, *Deep Learning: Foundations and Concepts*, https://issuu.com/cmb321/docs/deep_learning_ebook.

[3] J. LEI BA, J. RYAN KIROS, G. E. HINTON, *Layer Normalization*, https://arxiv.org/pdf/1607.06450.pdf.

[4] C. M. BRAGAGNINI, *Traducción Automática del Español al Inglés usando Redes Neuronales Profundas con Información Conceptual de Secuencias*, https://repositorio.ucsp.edu.pe/backend/api/core/bitstreams/c2deee4d-2672-47c1-93f9-2fd935628bbf/content.

[5] E. V. CUEVAS, J. V. OSUNA, D. A. OLIVA, M. A. DÍAZ, *Optimización: Algoritmos programados con MATLAB*.

[6] L. COELLO, L. CASAS, O. L. PÉREZ, Y. CABALLERO, *Redes neuronales artificiales en la producción de tecnología educativa para la enseñanza de la diagonalización*, https://dialnet.unirioja.es/servlet/articulo?codigo=5104742.

[7] O. CAMPESATO, *Transformer, BERT and GPT: Including ChatGPT and Prompt Engineering*, https://www.google.es/books/edition/Transformer_BERT_and_GPT3/O4_kEAAAQBAJ?hl=es&gbpv=1&dq=Transformer+IA&printsec=frontcover.

[8] X. CHEN, C. LIANG, D. HUANG, E. REAL, K. WANG, H. PHAM, X. DONG, T. LUONG, C. HSIEH, Y. LU, Q. V. LE, *Symbolic Discovery of Optimization Algorithms*, https://proceedings.neurips.cc/paper_files/paper/2023/file/9a39b4925e35cf447ccba8757137d84f-Paper-Conference.pdf.

[9] C. CAROLLO, *Apuntes Regresión lineal - Universidad Santiago de Compostela*, http://eio.usc.es/eipc1/BASE/BASEMASTER/FORMULARIOS-PHP-DPTO/MATERIALES/Mat_50140116_Regr_%20simple_2011_12.pdf.

[10] G. CYBENKO, *Approximation by Superpositions of a Sigmoidal Function*, https://web.njit.edu/~usman/courses/cs675_fall18/10.1.1.441.7873.pdf.

[11] M. V. CHACÓN, *Estudio de la reducción del sobreajuste en arquitecturas de redes neuronales residuales ResNet en un escenario de clasificación de patrones*, https://repositorio.unal.edu.co/bitstream/handle/unal/84211/1085325637.2023.pdf?sequence=2&isAllowed=y.

[12] J. DAGNINO, *Regresión lineal*, https://www.sachile.cl/upfiles/revistas/54e63943b5d69_14_regresion-2-2014_edit.pdf.

[13] M. D. FIUZA, J. C.. RODRÍGUEZ, *La regresión logística: una herramienta versátil*, https://www.revistanefrologia.com/es-la-regresion-logistica-una-herramienta-articulo-X0211699500035664.

[14] A. JAISWAL, A. RAMESH BABU, M. ZAKI ZADEH, D. BANERJEE, F. MAKEDON, *A Survey on Contrastive Self-Supervised Learning*, https://www.mdpi.com/2227-7080/9/1/2.

[15] G. HUANG, Y. SUN, Z. LIU, D. SEDRA, K. Q. WEINBERGER, *Deep Networks with Stochastic Depth*, https://arxiv.org/pdf/1603.09382.

[16] IBM, https://www.ibm.com/.

[17] A. KRIZHEVSKY, I. SUTSKEVER, G. E. HILTON, *ImageNet classification with deep convolutional neural networks*, https://dl.acm.org/doi/10.1145/3065386.

[18] S. KUBLIK, S. SABOO, *GPT-3 The ultimate guide to building NLP products with OpenAi API*, http://www.lcace.org/Media/GPT-3.pdf.

[19] M. KOWSHER, A. AS, N. J. PROTTASHA, M. S. AREFIN, P. K. DHAR, T. KOSHIBA, *Bangla-BERT: Transformer-Based Efficient Model for Transfer Learning and Language Understanding*, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9852438.

[20] I. LOSHCHILOV, F. HUTTER, *Decoupled weight decay regularization*, https://arxiv.org/pdf/1711.05101.pdf.

[21] LIBRERIA PYTORCH, https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html.

[22] Y. LECUN, Y. BENGIO, G. HINTON, *Deep Learning*, https://www.nature.com/articles/nature14539.

[23] G. LARSSON, M. MAIRE, G. SHAKHNAROVICH, *FRACTALNET: ULTRA-DEEP NEURAL NETWORKS WITHOUT RESIDUALS*, https://arxiv.org/pdf/1605.07648.

[24] LIBRERIA PYTORCH, https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html.

[25] S. J.D. PRINCE, *Understanding Deep Learning*, https://udlbook.github.io/udlbook/.

[26] I. PÉREZ , M.E. GEGÚNDEZ , *Deep Learning*, https://www.google.es/books/edition/DEEP_LEARNING/kzsvEAAAQBAJ?hl=es&gbpv=1&dq=optimizaci%C3%B3n+por+gradiente&pg=PA117&printsec=frontcover, 50-54.

[27] S. RUSSELL, P. NORVIG, *Inteligencia Artificial. Un enfoque moderno.*, https://luismejias21.wordpress.com/wp-content/uploads/2017/09/inteligencia-artificial-un-enfoque-moderno-stuart-j-russell.pdf.

[28] D. ROTHMAN, *Transformers for Natural Language Procesing* 21-54.

[29] R. K. SRIVASTAVA, K. GREFF, J. SCHMIDHUBER, *Highway Networks*, https://arxiv.org/pdf/1505.00387.pdf.

[30] K. SHAH , R. W. N. NG, F. BOUGARES, L. SPECIA, *Investigating Continuous Space Language Models for Machine Translation Quality Estimation*, https://aclanthology.org/D15-1125.pdf.

[31] L. TUNSTALL, L. VON WERRA, T. WOLF *Natural Language Processing with Transformers* 57-84.

[32] J. A. TORREJÓN, J. J. BARDALES, W. J. FELIPE *El Razonamiento Matemático de modelos Transformer*.

[33] A. VASWANI, N. SHAZEER, N. PARMAR, J. USZKOREIT, L. JONES, A. N. GOMEZ, L. KAISER, I. POLOSUKHIN *Attention Is All You Need*, https://arxiv.org/pdf/1706.03762.