Sistemas de Inteligencia Artificial

Transformers

Eugenia Sol Piñeiro

Octubre 2021

Índice general

0.1. Introducción

El Paper *'Attention is All You Need'* [6] publicado en el año 2017 introduce una arquitectura de *Encoders* y *Decoders* basada en un mecanismo de atención, a la que denominaron *'Transformer'*.

Un Transformer es una red neuronal cuyo principal objetivo es transformar una secuencia en otra.

A diferencia de otros tipos redes como las Redes Recurrentes (RNN), las Redes Convolucionales o Long Short-Term Memory (LSTM) [3], permite ejecutar tareas de forma paralela reduciendo el tiempo requerido para entrenar.

A su vez, puede recordar viejas conexiones en secuencias de gran tamaño evitando el problema del gradiente que va desapareciendo o crece demasiado. *(vanishing/exploding gradients).*[2]

0.2. Arquitectura

La arquitectura de un Transformer consiste en un conjunto de *Encoders* y *Decoders* que interactúan entre sí para a partir de una secuencia (input) generar otra (output).

A fines de comprender mejor la arquitectura, supongamos que se quiere traducir la secuencia de palabras *'Soy un estudiante'* del español al inglés, esperando como resultado *'I am a student'*. (ver Figura 1)

Cabe destacar que se puede contar con tantos *Encoders* y *Decoders* apilados como se desee, considerando que la cantidad debe ser la misma tanto para el *enconding* como para el *decoding*.

El output de cada *Encoder* será tomado como input del próximo, hasta llegar al último *Encoder* cuyo output será tomado por todos los *Decoders*, junto con el output del *Decoder* anterior. Todos los *Encoders* tienen una estructura idéntica, aunque difieren en los pesos. (ver Figura 2)

Los inputs del *Encoder* primero pasan por una capa denominada *Self-Attention Layer* sobre la cual se profundizará más adelante. Los outputs de dicha capa ingresan por una red neuronal *Feed-Forward* que luego serán to-

mados como input de los *Decoders*.

Cada *Decoder* también posee una capa de *Self-Attention* y una red neuronal *Feed-Forward* pero además agrega una capa en el medio denominada *Encoder-Decoder Attention*. Esta capa le ayuda al *Encoder* a enfocarse en partes relevantes de la sentencia inicial. (ver Figura 3)

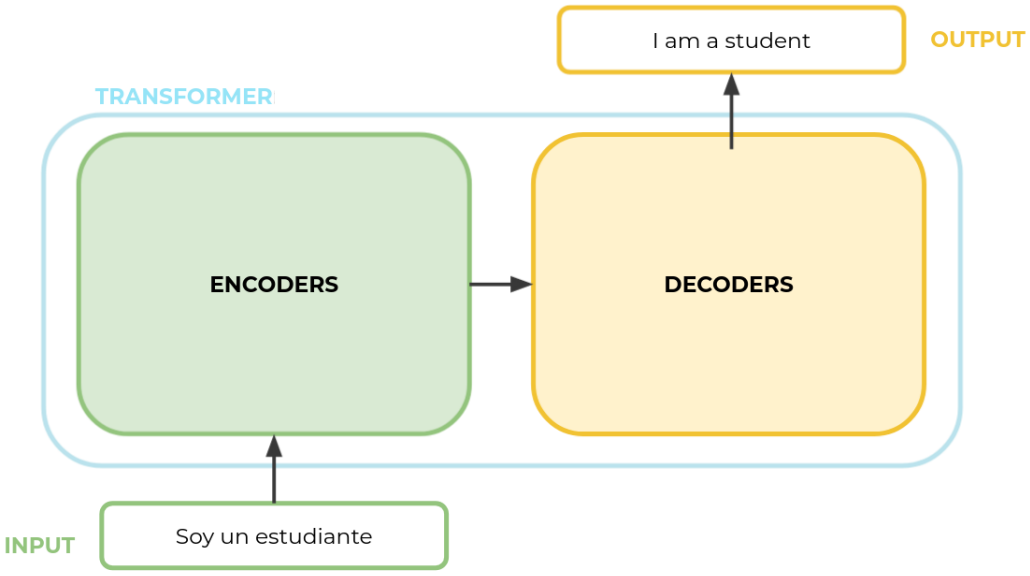

Figura 1: Diagrama de Arquitectura Básico de un Transformer.

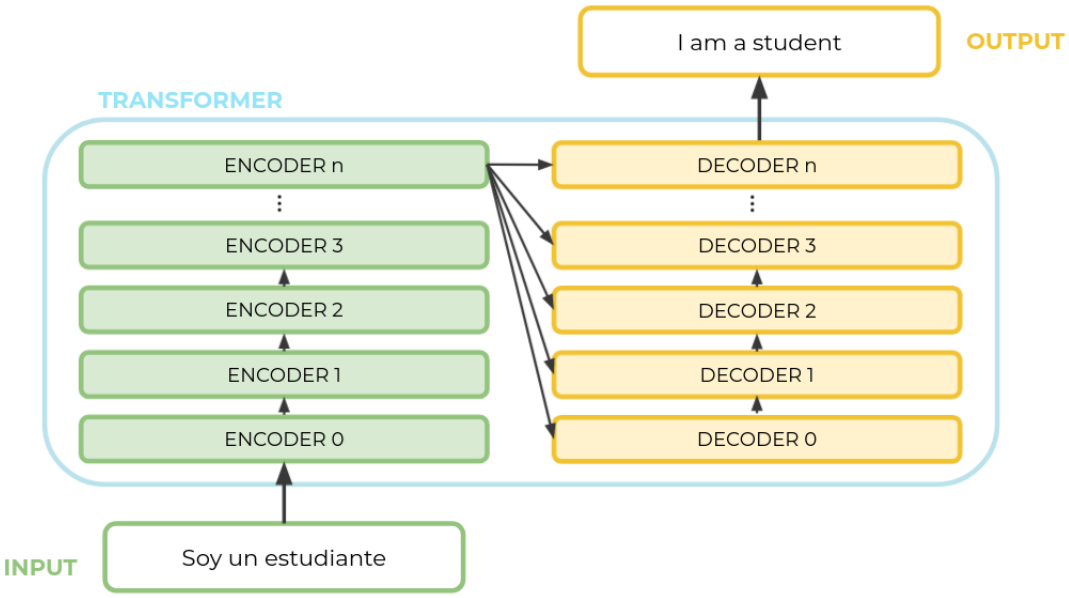

Figura 2: Propagación de inputs y outputs con múltiples Enconders y Decoders

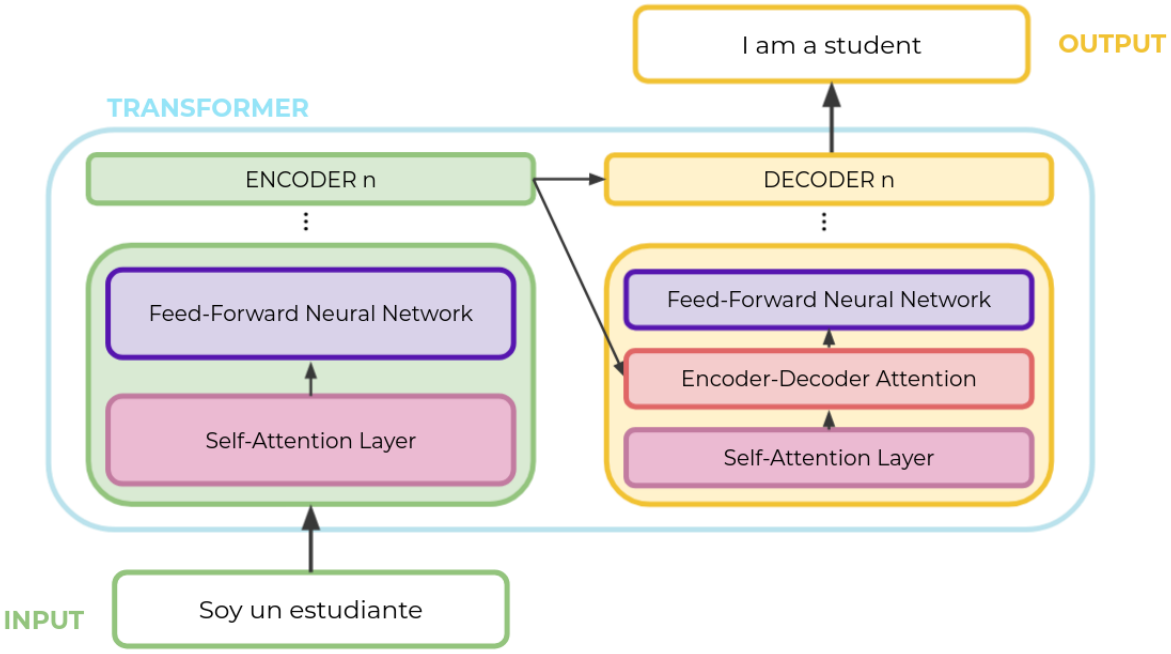

Figura 3: Detalle de las capas internas de cada Encoder y cada Decoder.

0.2.1. Self-Attention Layer

Se trata de una capa interna que poseen todos los *Encoders* y *Decoders* cuya función es tener en cuentra otras palabras de la secuencia mientras se está analizando una en particular.

En el caso de los *Encoders*, esta capa le permite poner el foco en cualquier otra palabra mientras que un *Decoder* únicamente puede analizar palabras previas en la secuencia del output.

Por ejemplo, se tiene la secuencia *'The animal didn't cross the street because* ***it*** *was tired'*. Nos podríamos preguntar, ¿la palabra '**it**' hace referencia a la palabra *'animal'* o a la palabra *'street'*?.

Si bien para una persona con un mínimo conocimiento del idioma puede responder con facilidad la pregunta, un Transformer necesita de la capa de *Self-Attention*.

Cálculo de Self-Attention

Para lograr su objetivo esta capa debe realizar una serie de operaciones que veremos a continuación.

Antes de ingresar a la capa de *Self-Attention* el input pasa por una serie de transformaciones.

En primer lugar, se realiza el proceso de ***Tokenization***, se toma el input y se lo divide en secuencias más pequeñas llamadas *tokens*.

Luego, cada *token* es representado en un vector de números reales de dimensión 512. A este proceso se lo denomina ***Embedding***. (ver Figura 4)[1]

Existen diversas formas de calcular los embeddings, entre ellas se destacan los algortimos *GloVe* y *word2vec*. Este último tiene dos variantes *SkipGram* y *CBOW*.[5]

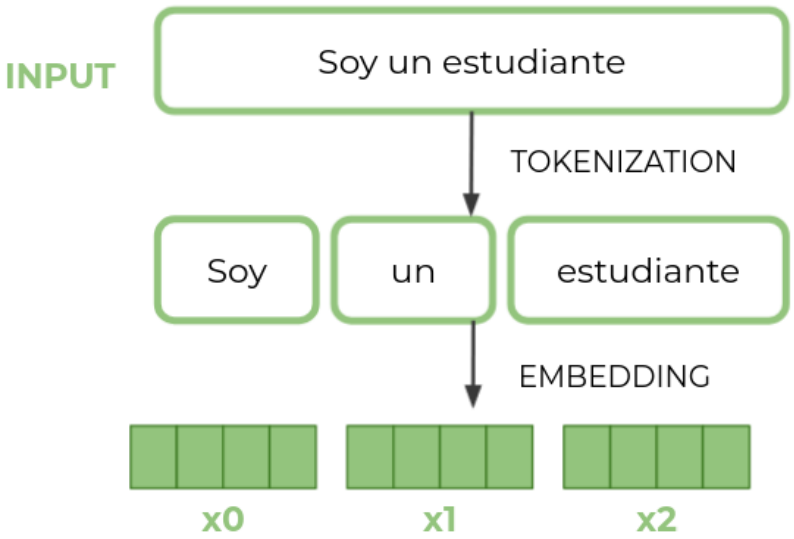

Figura 4: Transformación del input para ingresar a la capa de Self-Attention.

Una vez obtenidos los *embeddings*, llamémoslos x_i, se procede a hacer el cálculo de *Self-Attention*.

[1]No necesariamente el input debe ser una secuencia de palabras, y en caso de serlo no necesariamente se divide por el caracter 'espacio'. La figura es a fin ilustrativo.

Por cada uno de los *embeddings* x_i, se calculan tres vectores llamados *query* q_i, *key* k_i y *value* v_i, multiplicando por otras tres matices (WQ, WK y WV) que poseen los pesos de atención (*Attention Weights*). (ver Figura 5)

Luego, se calcula un *score* s_i por cada *embedding*. El *score* determina cuánta atención se debe poner en otras partes de la sentencia mientras se está analizando un token en particular. Para ello, se computa el producto punto entre q y el vector k de cada *embedding*.

Los *scores* se dividen por la raíz de la dimensión del vector k (d_i) y se normalizan con el método *SoftMax* [8] obteniendo un conjunto de probabilidades p_i.

Por último, se multiplica cada vector v_i por lo obtenido en la función de Softmax y se se suman obteniendo el output de la capa de Self-Attention para cada token.(ver Figura 6)

De esta forma, el input pasará por la capa de *Self-Attention* una vez transformado, se calcularán los vectores de salida, llamémoslos z_i, que luego pasarán por la red neuronal *Feed-Forward*.

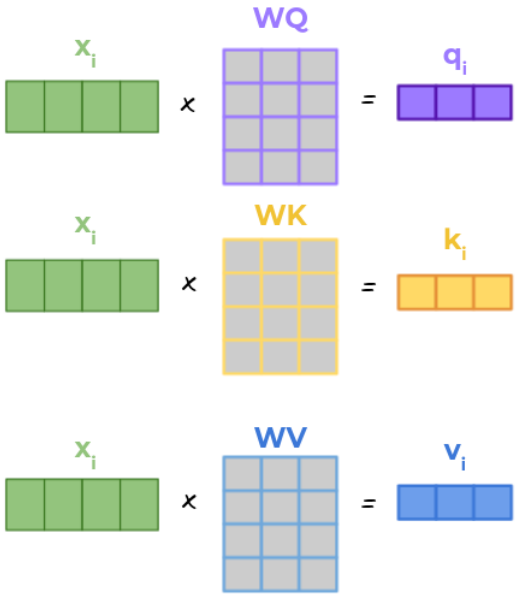

Figura 5: Multiplicación de cada embedding (x_i) por los pesos de Self Attention para obtener los vectores query (q_i), key (k_i) y value (v_i)

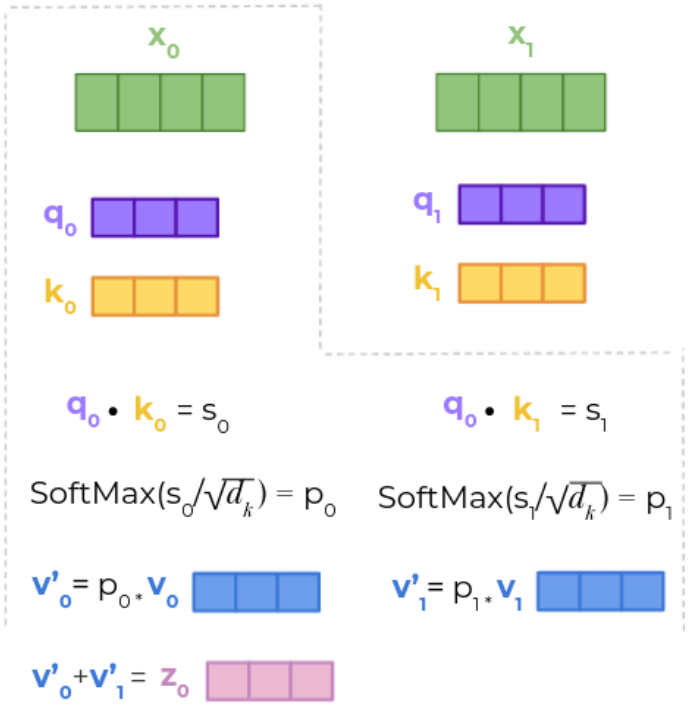

Figura 6: Cálculo de Self Attention para el primer emebdding x_0.

A la hora de implementar un Transformer, este cálculo se realiza de forma matricial para realizar un procesamiento más rápido. Se unen los *embeddings* en una matriz X. Los vectores q, k y v pasan a ser tres matrices también llamadas *query* (Q), *key* (K) y *value* (V)(ver Figura 7)

En tercer lugar, se multiplican K y Q para obtener los (*scores*). Los mismos serán divididos por la la raíz de la dimensión de los vectores fila de la matriz K (d_k) y se normalizan usando *SoftMax*.

Por último, se multiplica por la matriz V para mantener intactos los valores del token en el que nos queremos enfocar y descartando los irrelevantes.

Unificando los pasos se obtiene la siguiente fórmula:

$$Attention(Q, K, V) = SoftMax(\frac{QK'}{\sqrt{d_k}}) * V \tag{1}$$

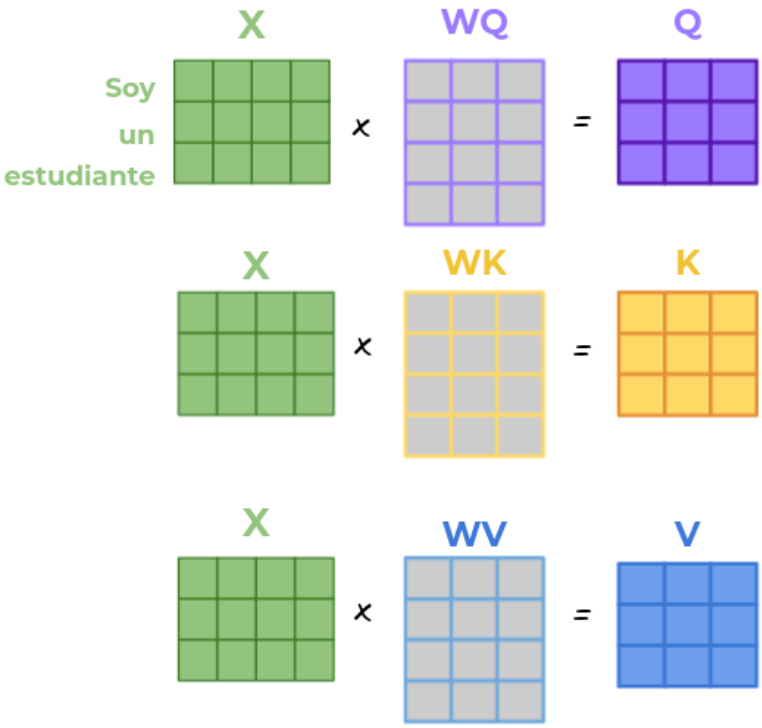

Figura 7: Matrices utilizadas para el cálculo de Self-Attention

Cabe destacar, que una vez transformados, cada token sigue su propio camino en el *Encoder*, pudiendose ejecutar en el paralelo, pasando por redes *Feed-Forward* idénticas. Estos caminos tienen ciertas dependencias en el capa de *Self-Attention* pero no en la red *Feed-Forward*. (ver Figura 8)

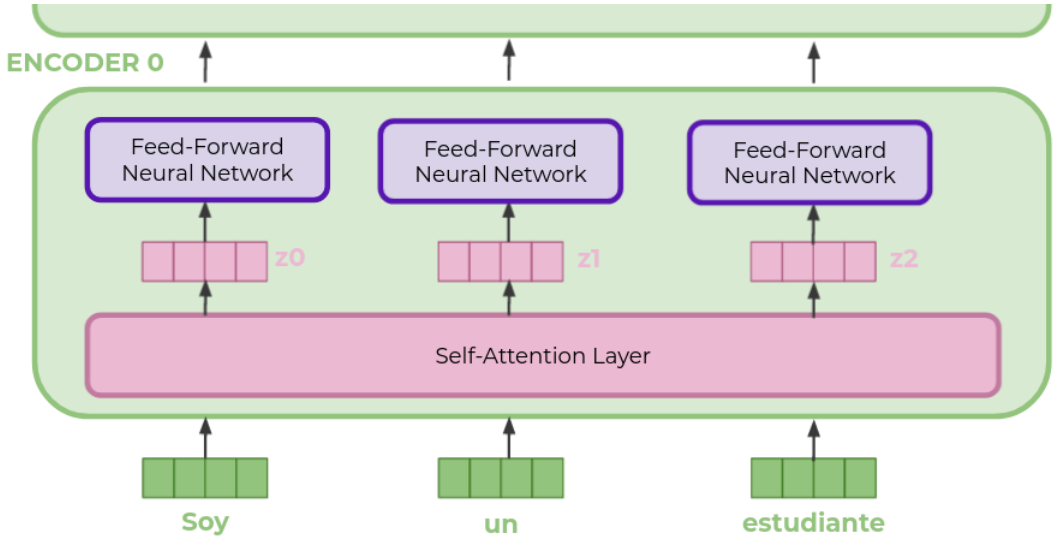

Figura 8: outputs z_i ingresando a múltiples redes Feed-Forward en paralelo.

Multi-Headed Attention

Con el objetivo de mejorar la performance de la capa de *Self-Attention* se puede utilizar un mecanismo denominado *Multi-Headed Attention*. El mismo consiste en tener múltiples matrices K, Q y V permitiendo al encoder enfocarse en distintas posiciones de la secuencia simultáneamente.

Volvamos al ejemplo anterior en el que teníamos la siguiente secuencia: *'The animal didn't cross the street because* ***it*** *was tired'*. A medida que se agregan capas de *Self-Attention* se puede notar en qué tokens se está enfocando cada token en particular. (ver Figura 9). Lo interesante es ver cómo la palabra *it* se está enfocando en su mayoría en la palabara *animal*, y al agregar otra capa se puede notar que a su vez su foco está en la palabra *tired*.

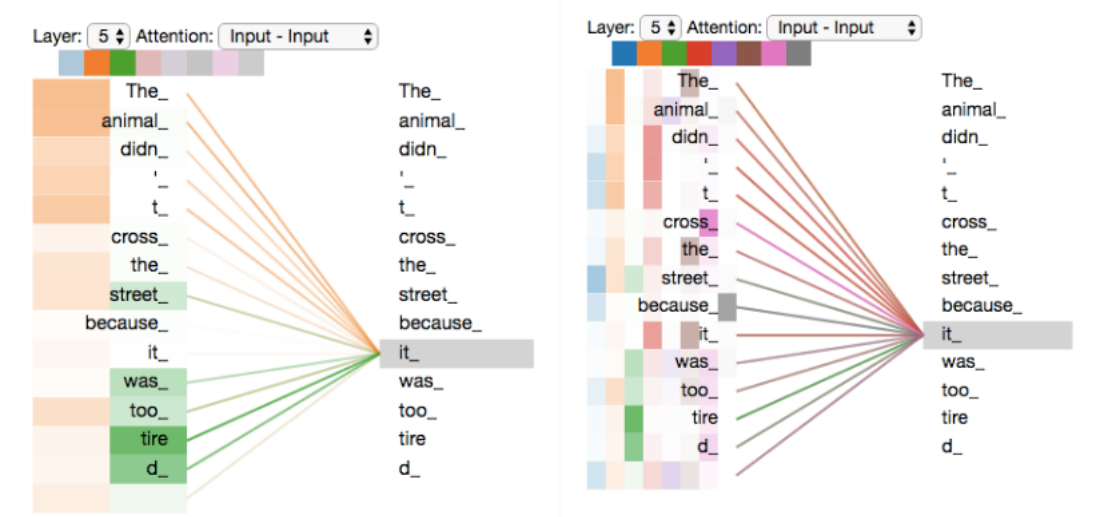

Figura 9: Multi-Headed Attention Ver visualización en Tensor2Tensor

0.2.2. Positional Encoding

A medida que cada token de la secuencia atraviesa cada una de las capas del Transformer, este último no lleva un registro de cuál es la posición de los *tokens* en la secuencia. Continuando con el ejemplo de la traducción de una oración, tener en cuenta el orden de las palabras es una parte esencial de un idioma.

Una posible solución es agregar información a cada palabra sobre su posición en la oración. A esta información se la llama *Positional Encoding*, un conjunto de vectores p_i que se le suma a los *embeddings*. (ver Figura 10)

En principio, se podría asignar un número entero de forma lineal a cada *token*, es decir, al primero le correspondería el número 0, al segundo el 1 y así sucesivamente. El problema es que los valores podrían volverse muy grandes y además no necesariamente se entrena la red con secuencias de igual longitud que en el aprendizaje. La red debería ser capaz de generalizar secuencias de mayor longitud sin mucha dificultad.

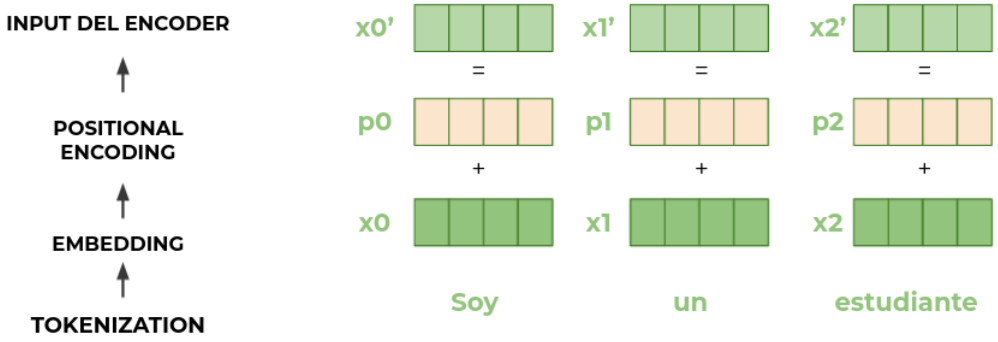

Figura 10: Transformación del input teniendo en cuenta el orden de los tokens

Si bien hay diversas formas de abordar este inconveniente, en el paper de *Attention Is All You Need*[1] se optó por utilizar las siguientes funciones sinusoidales para obtener el vector de posición del *token* i:

$$PE(pos, 2i) = sen(\frac{pos}{10000^{\frac{2i}{d}}}) \tag{2}$$

$$PE(pos, 2i+1) = cos(\frac{pos}{10000^{\frac{2i}{d}}}) \tag{3}$$

Donde:

- pos es la posición del token en la secuencia.
- i es el índice del vector de posición.
- d es la dimensión de un *embedding*, la cual será la misma que un vector de posición p_i.

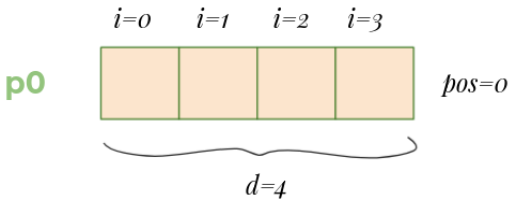

Figura 11: Referencias de las fórmulas sinusoidales

0.2.3. Linear y Softmax

Una vez que los tokens llegaron hasta el último *Decoder*, se obtiene como output un vector de números decimales. Pero, en realidad la salida esperada es una palabra.

Para transformar ese vector de números, se agregan dos capas: ***Linear*** y ***Softmax***. La primera se trata de una Red Neuronal *Fully-Connected* (FCNN) [7] que proyecta el output del último *Decoder* en un vector de mayor dimensión llamado *logits*. Cada uno de los elementos de este vector se corresponde con el puntaje de un token en particular.

Luego, la capa de *Softmax* convierte los puntajes en probabilidades. Se elige el índice cuyo valor posea la mayor probabilidad y la palabra asociada a esa probabilidad resulta como output. (Ver Figura 12)

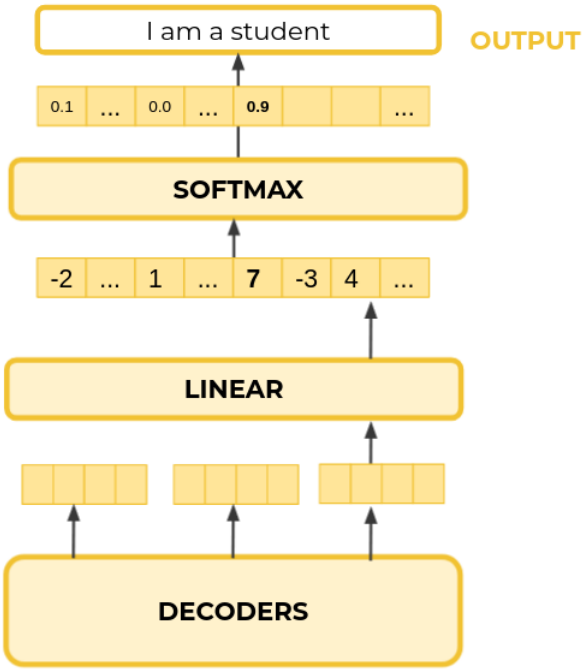

Figura 12: Capas Linear y Softmax

Combinando todos los conceptos mencionados previamente, la arquitectura de un Transformer se puede resumir de la siguiente forma: (ver Figura 13)

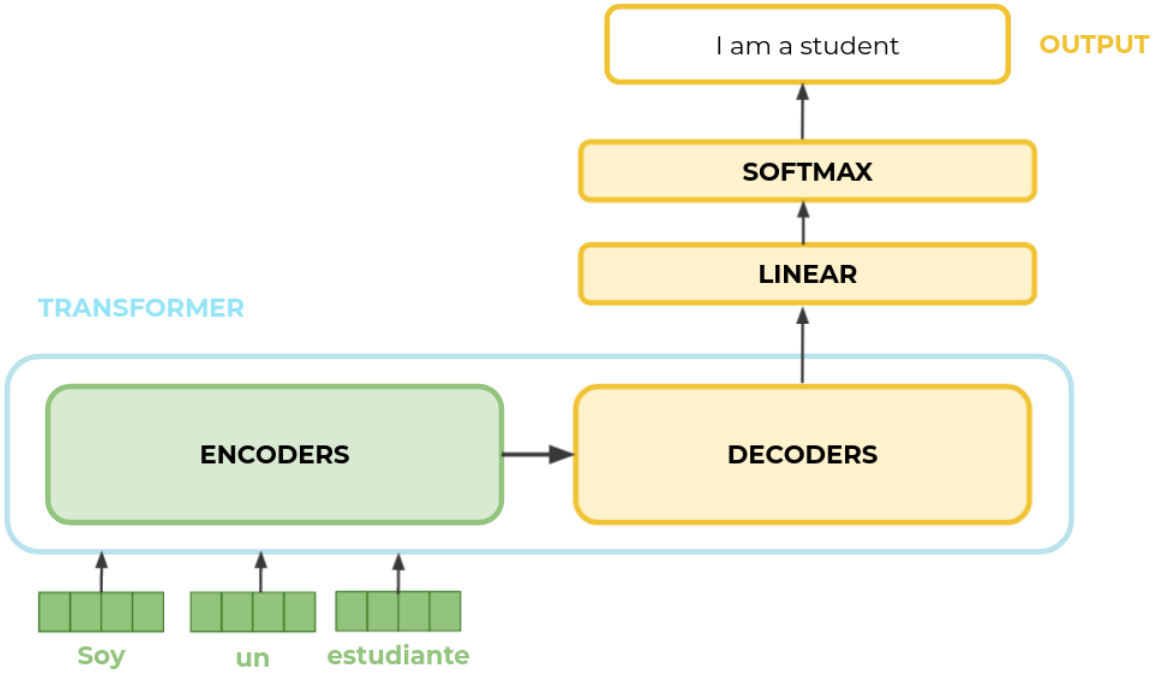

Figura 13: Arquitectura de un Transformer

0.3. Entrenamiento

Supongamos que se desea traducir la palabra *'gracias'* a *'thanks*. Se posee un vocabulario de salida con múltiples palabras en el idioma del output deseado $v = ['a', 'am', 'i', 'thanks', 'student', 'EOF']$, siendo 'EOF' un *token* que indica la finalización de la secuencia (End Of File).

Se incializan los pesos de forma aleatoria, entonces el modelo sin entrenar devuelve una distribución de probabilidades con valores arbitrarios para cada token. Estas probabilidades se pueden comparar con el método de *Cross-Entropy*.

En una primera iteración, se obtendrán probabilidades más lejanas a la deseada y luego por *backpropagation* nos vamos acercando al output deseado.(ver Figura 14)

Figura 14: Ejemplo de Entrenamiento para traducir la palabra 'gracias' y obtener 'thanks'.

0.4. Aplicaciones

Si bien a lo largo de la explicación se utilizó un ejemplo sobre traducción de secuencias de un idioma para mejor comprensión, los Transformers tienen diversas aplicaciones:

- Comprensión, generación y traducción de textos: OpenAI Language Model
- Predecir qué va a suceder: Intellisense en Visual Studio Code
- Juego de Estrategia Real-Time: StarCraft II
- Detección de anomalías [4]
- Reconocimiento de Imágenes [1]

0.5. Transformers más conocidos

A lo largo de los años fueron surgiendo diversas arquitecturas basadas en el Transformer básico explicado anteriormente. Entre ellas podemos encontrar:

- BERT
- DisitilBERT
- T5
- GPT-2

Bibliografía

[1] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[2] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Hengyu Meng, Yuxuan Zhang, Yuanxiang Li, and Honghua Zhao. Spacecraft anomaly detection via transformer reconstruction error. In *International Conference on Aerospace System Science and Engineering*, pages 351–362. Springer, 2019.

[5] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[7] Yuchen Zhang, Jason Lee, Martin Wainwright, and Michael I Jordan. On the learnability of fully-connected neural networks. In *Artificial Intelligence and Statistics*, pages 83–91. PMLR, 2017.

[8] Zhen Zhao, Ashley Kleinhans, Gursharan Sandhu, Ishan Patel, and KP Unnikrishnan. Capsule networks with max-min normalization. *arXiv preprint arXiv:1903.09662*, 2019.