

# Attention

# Concepto

**Attention** es un mecanismo muy potente en NLP (en deep learning en general) para **ponderar la información según su relevancia en el contexto** en cada cadena de texto o cadena de secuencias

De esta forma,

**Attention** permite ignorar **información irrelevante** y centrarnos en la **información importante**

Desde un punto de vista práctico, acabaremos creando una **Attention Matrix** que pondere cada palabra de una secuencia o cadena de secuencias



# Motivación

"Ana va al cine a ver una nueva ----"

Por ejemplo, dada la frase anterior, ¿cuál sería la siguiente palabra?

película  
comedia  
película de risa  
película francesa

tienda  
exposición  
partida de ajedrez  
obra teatral

Por cómo está construida la frase, podríamos ir a *“un sitio”* ver cualquiera de las opciones propuestas. Sin embargo, sabemos que en el cine, lo que se suele ver son películas. Por tanto, ese contexto que te proporciona la palabra cine, ya permite aumentar las probabilidades de que vaya a ver unas cosas respecto de otras.



# Motivación

Los Language Models (LMs) están entrenados para predecir la próxima palabra basándose en el contexto de las palabras anteriores. Sin embargo, para hacer predicciones precisas, los LMs necesitan entender la relación entre las palabras en la oración. Este es el objetivo del mecanismo de atención:

**Ayudar** al LM a enfocarse en las palabras más relevantes para ese contexto para hacer predicciones

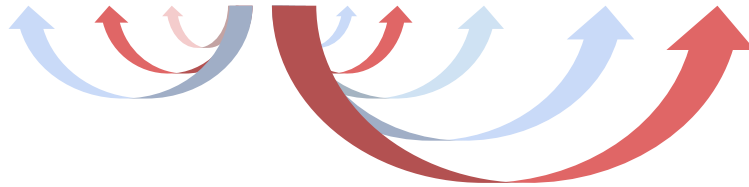
En la oración, cada palabra tiene que prestar atención a otras palabras con diferentes grados de atención, basado en el contexto y la similitud de esas palabras.

- Por ejemplo, la palabra *cine* y *ver* tienen una relación más fuerte en este contexto, lo que puede ayudar a predecir palabras relacionadas con películas en el espacio en blanco.



# Relación entre palabras

Ana va al **cine** a ver una nueva -----



Alto valor de relación

película  
comedia  
película de risa  
película francesa

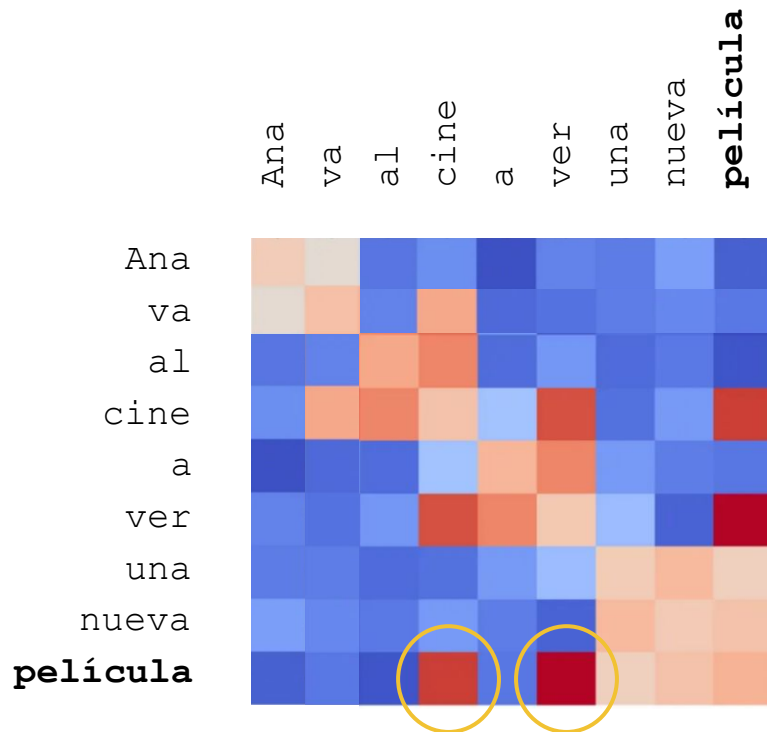
Ana va al **cine** a ver una nueva -----



Bajo valor de relación

tienda  
exposición  
partida de ajedrez  
obra teatral

# Relación entre palabras



Al final, de lo que se trata es de construir una **Context-similarity matrix**

# Attention is all you need

El mecanismo de Attention primeramente abordado como una parte de RNN sobre arquitecturas *encoder-decoder* (Bahdanau et al. 2015, Sutskever et al. 2014).

Sin embargo, fue en 2017 con el artículo *Attention is all you need* cuando la fue introducido con mayor éxito.

- Primera vez introducido con MLP (en lugar de RNN)
- Primera vez con una capa Scaled (que reducía el problema de gradient vanishing)

Este mecanismo funcionaba tan bien, que dio lugar al nacimiento de los **Transformers** (con **Bert** como primera arquitectura Transformer\*)

---

## Attention Is All You Need

---

Ashish Vaswani\*  
Google Brain  
avaswani@google.com

Noam Shazeer\*  
Google Brain  
noam@google.com

Niki Parmar\*  
Google Research  
nikip@google.com

Jakob Uszkoreit\*  
Google Research  
usz@google.com

Llion Jones\*  
Google Research  
llion@google.com

Aidan N. Gomez\*<sup>†</sup>  
University of Toronto  
aidan@cs.toronto.edu

Lukasz Kaiser\*  
Google Brain  
lukaszkaizer@google.com

Illia Polosukhin\*<sup>‡</sup>  
illia.polosukhin@gmail.com

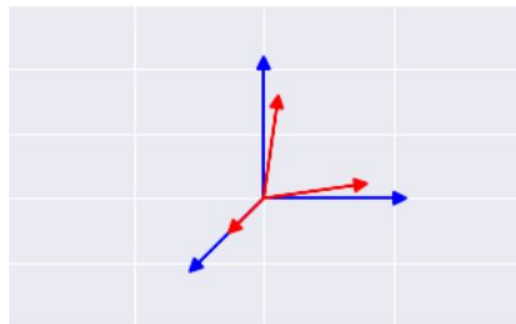


# Mecanismos de atención

Los mecanismos de atención toman un conjunto de vectores como entrada y tienen la tarea de producir otro conjunto conjunto de vectores como salida.

Lo que se genere variará según el tipo de mecanismo empleado.

- En el caso del **hard attention**, cada vector resultante se enfoca exclusivamente en un solo vector de entrada, el que ocupa la misma posición que él.
- El de **soft attention** crea vectores que resultan de la mezcla de todos los vectores de entrada.
- El de **self-attention** es una variante de soft attention donde la relevancia de cada vector de entrada se determina según su parecido con los demás vectores ingresados.

 $X$  $AX$ 

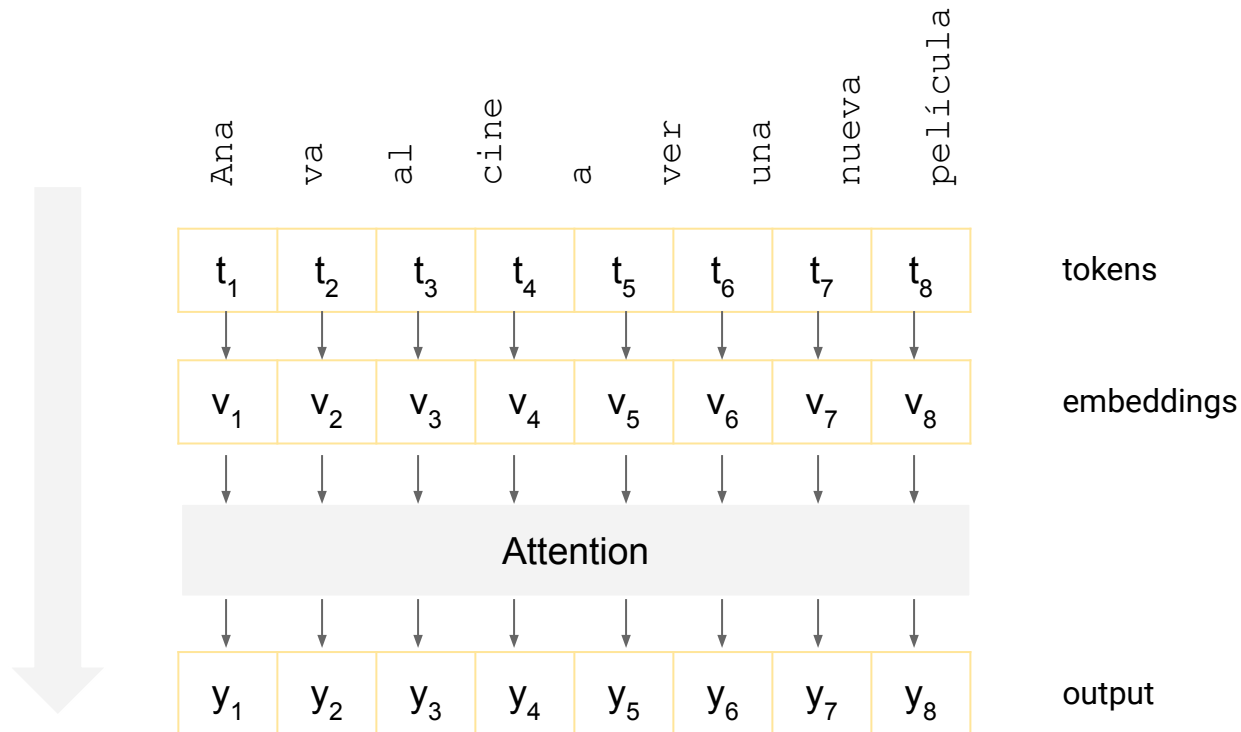
dot product  
(MatMul)





# Mecanismo de atención

¿Cómo funciona el mecanismo de atención a alto nivel?



# Scaled dot-product self Attention

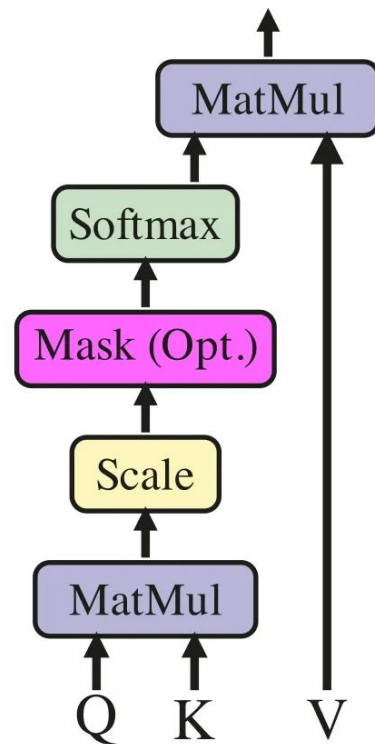
Mientras que el *dot-product Attention* existía antes de este artículo, la introducción de un **factor de escala** mejoró significativamente el rendimiento del mecanismo.

Antes de este artículo, la atención se utilizaba principalmente en redes recurrentes.

El *scaled dot-product Attention* toma tres conjuntos de vectores como entrada:

**Queries (Q)**, **Keys (K)** y **Values (V)**:

- **Query**: cuán relevantes son otros elementos (como palabras en una oración) respecto al elemento actual.
- **Key**: cuando una query busca la relevancia o atención de otros elementos, se compara con las keys para determinar qué tan alineados o similares son.
- **Values**: son los que realmente se agregan o se toman en cuenta. Representan la información real que se quiere extraer de la secuencia de referencia, ponderada por la importancia calculada en el proceso de atención

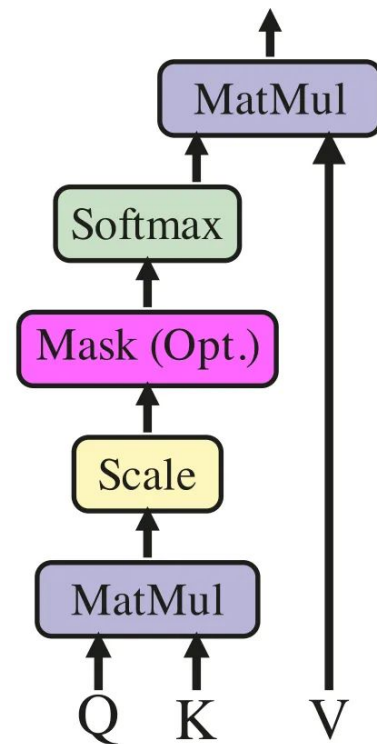


# Scaled dot-product self Attention

Primero, se calculan las puntuaciones de atención mediante el producto punto de Queries (Q) con Keys (K), y luego se escalan por un factor derivado de la dimensión de las keys ( $d_k$ ):

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

- El escalado es crucial. Sin este término, los productos punto podrían crecer muy grandes en magnitud, empujando la función softmax hacia regiones donde tiene gradientes muy pequeños (es decir, en el problema del gradiente que desaparece).
- A las puntuaciones resultantes se les aplica la función softmax para obtener los pesos de atención, asegurando que sumen 1 y puedan interpretarse como probabilidades.
- Finalmente, los pesos de atención se multiplican por el conjunto de vectores Values, para obtener el resultado final del mecanismo de atención.



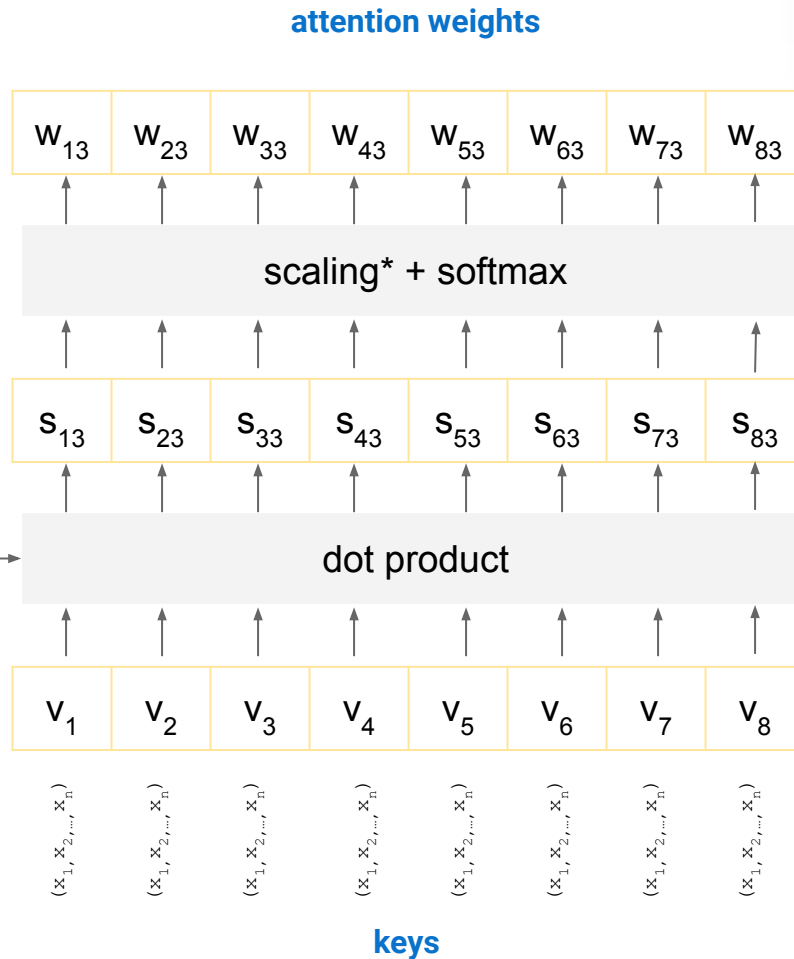
# Stepwise

¿Cómo funciona el mecanismo de atención a bajo nivel?

query

$(x_1, x_2, \dots, x_n)$

$v_3$



keys

\*  $\text{sqrt}(8)$  -> dimensión de las keys



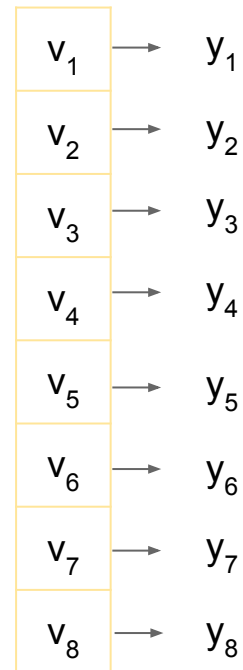
# Stepwise

attention matrix

$w_{11}$	$w_{21}$	$w_{31}$	$w_{41}$	$w_{51}$	$w_{61}$	$w_{71}$	$w_{81}$
$w_{13}$	$w_{23}$	$w_{33}$	$w_{43}$	$w_{53}$	$w_{63}$	$w_{73}$	$w_{83}$
...							
$w_{18}$	$w_{28}$	$w_{38}$	$w_{48}$	$w_{58}$	$w_{68}$	$w_{78}$	$w_{88}$

dot product

values

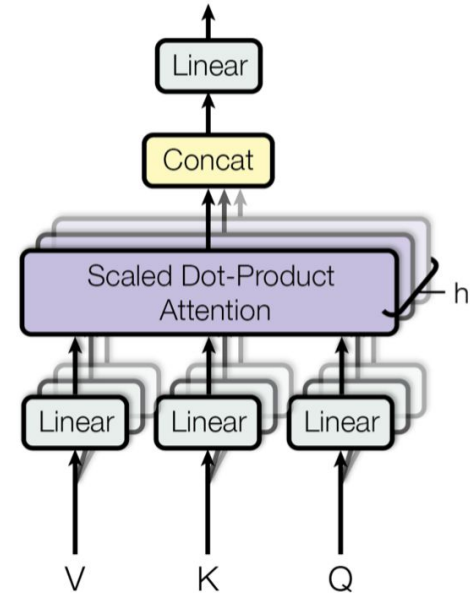


outputs

# Multihead Attention

En lugar de tener una sola "cabeza" de Scaled Dot-Product Attention analizando la secuencia de entrada una vez, Multihead Attention utiliza **múltiples cabezas** de atención que **operan en paralelo**.

Cada cabeza de atención aprende a **enfocarse en diferentes partes** de la secuencia de entrada, lo que permite al modelo capturar una gama más rica de relaciones contextuales.



# Keras Attention

## Attention layer

Attention class

```
keras.layers.Attention(  
    use_scale=False, score_mode="dot", dropout=0.0, seed=None, **kwargs  
)
```

Inputs are a list with 2 or 3 elements: 1. A **query** tensor of shape `(batch_size, Tq, dim)`. 2. A **value** tensor of shape `(batch_size, Tv, dim)`. 3. A optional **key** tensor of shape `(batch_size, Tv, dim)`. If none supplied, **value** will be used as a **key**.

The calculation follows the steps: 1. Calculate attention scores using **query** and **key** with shape `(batch_size, Tq, Tv)`. 2. Use scores to calculate a softmax distribution with shape `(batch_size, Tq, Tv)`. 3. Use the softmax distribution to create a linear combination of **value** with shape `(batch_size, Tq, dim)`.

## MultiHeadAttention layer

MultiHeadAttention class

```
keras.layers.MultiHeadAttention(  
    num_heads,  
    key_dim,  
    value_dim=None,  
    dropout=0.0,  
    use_bias=True,  
    output_shape=None,  
    attention_axes=None,  
    kernel_initializer="glorot_uniform"
```



# Ejemplo

```
input_text      = Input(shape=(max_length,))
embedding_layer = Embedding(vocab_size, embedding_dim, input_length=max_length)(input_text)
conv1d_layer    = Conv1D(filters, kernel_size, activation='relu')(embedding_layer)

attention_output = Attention(use_scale=True)([conv1d_layer, conv1d_layer, conv1d_layer])
flattened_output = Flatten()(attention_output)

dense_layer_1 = Dense(6, activation='relu')(flattened_output)
output_layer  = Dense(1, activation='sigmoid')(dense_layer_1)

model = Model(inputs=input_text, outputs=output_layer)
```

Q

K

V



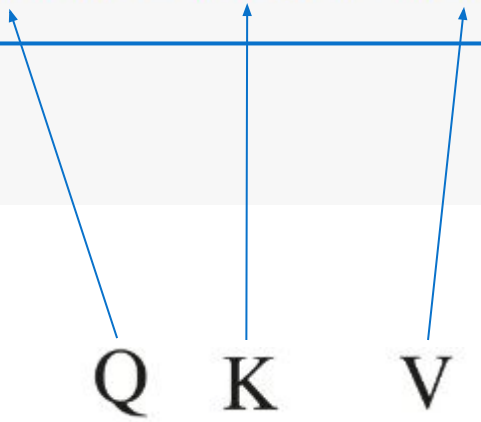
# Ejemplo

```
input_text          = Input(shape=(max_length,))
embedding_layer     = Embedding(vocab_size, embedding_dim, input_length=max_length)(input_text)
conv1d_layer        = Conv1D(filters, kernel_size, activation='relu')(embedding_layer)

multihead_attention_output = MultiHeadAttention(num_heads=5, key_dim=64)(conv1d_layer, conv1d_layer, conv1d_layer)
flattened_output      = Flatten()(multihead_attention_output)

dense_layer_1       = Dense(6, activation='relu')(flattened_output)
output_layer        = Dense(1, activation='sigmoid')(dense_layer_1)

model = Model(inputs=input_text, outputs=output_layer)
```



Q      K      V

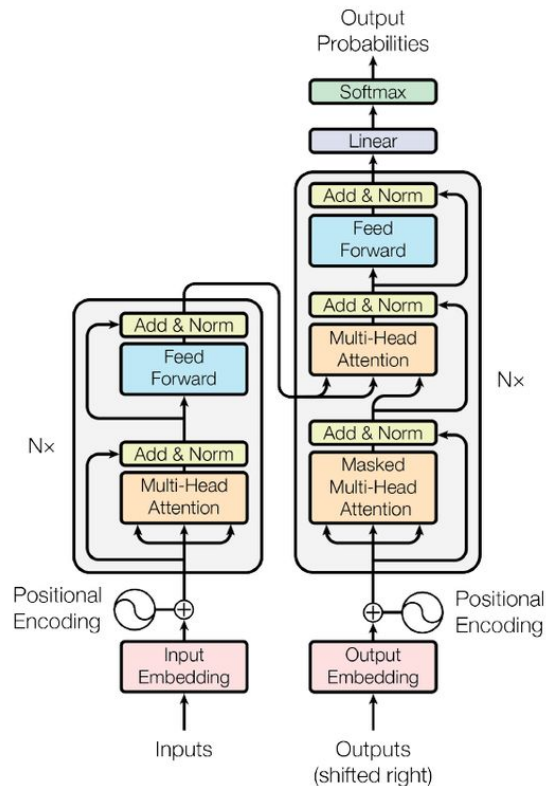
# Transformers

Los Transformers son un tipo de arquitectura de modelo que se ha vuelto fundamental en el procesamiento del lenguaje natural (NLP) y en varias otras áreas de la inteligencia artificial.

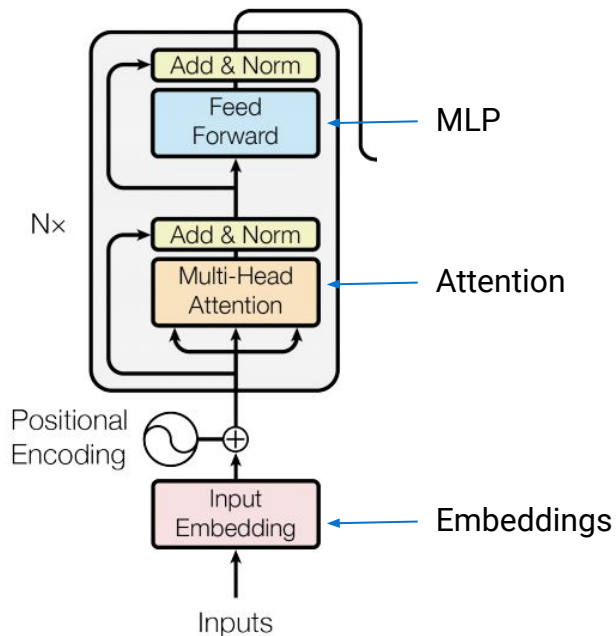
- Se basan en **mecanismos de atención**. Esto les permite ponderar la importancia de diferentes partes de las oraciones de manera dinámica, lo que mejora su capacidad para **entender el contexto** y las **relaciones entre palabras**.

Son capaces de aprender dependencias a largo plazo entre palabras en una oración.

- La arquitectura de los Transformers se divide principalmente en dos componentes: el **encoder** y el **decoder**.
- Además, es altamente **paralelizable**, lo que la hace muy eficiente para el entrenamiento en hardware moderno, como las **GPUs**.

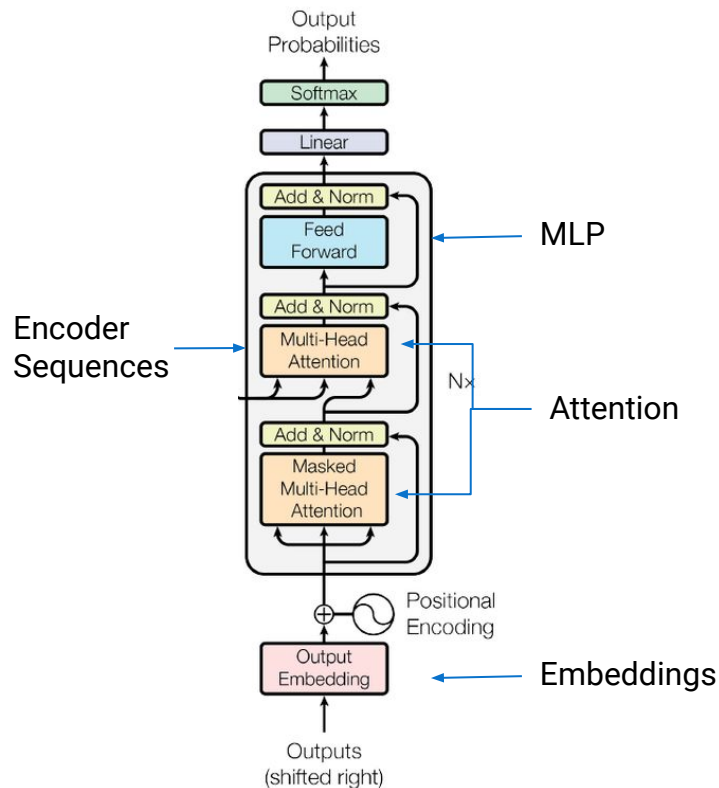


# Encoder



- El encoder transforma la secuencia de entrada en una secuencia de representaciones vectoriales.
- Estas representaciones contienen tanto la información original de la entrada como el contexto adicional recopilado a través del mecanismo de atención.
- Esto permite que cada elemento de la secuencia de salida, en el caso de que se utilice un decoder, tenga acceso a toda la secuencia de entrada al generar su salida.
- Algunos de los modelos más populares que usan solo encoders: BERT (Bidirectional Encoder Representations from Transformers).

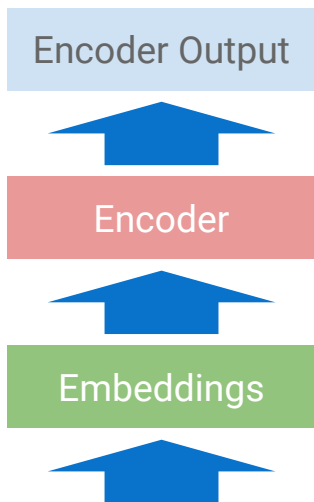
# Decoder



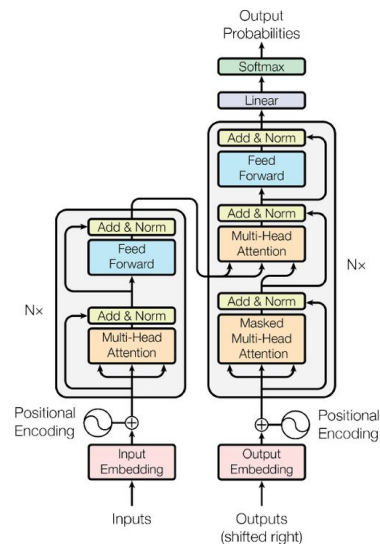
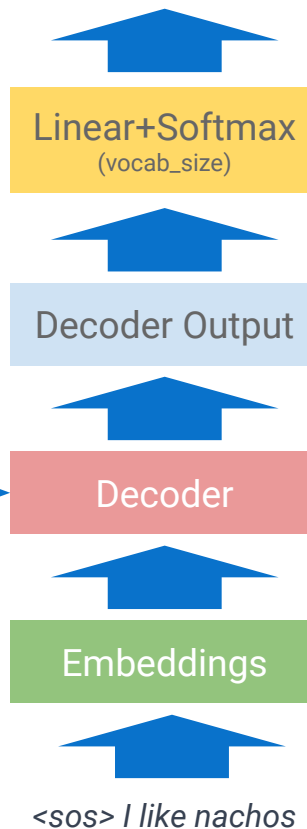
- El decoder opera de manera auto-regresiva, es decir, generando un elemento de la secuencia a la vez. Utiliza su salida hasta el momento como entrada para predecir el siguiente elemento de la secuencia.
- La interacción entre el encoder y el decoder a través del mecanismo de atención encoder-decoder permite que el decoder tenga en cuenta toda la información de la entrada al generar cada elemento de la secuencia de salida.
- Algunos de los modelos más populares que usan solo decoders: GPT (Generative Pretrained Transformer).

# Training

<sos> Me gustan los nachos <eos>



I like nachos <eos>

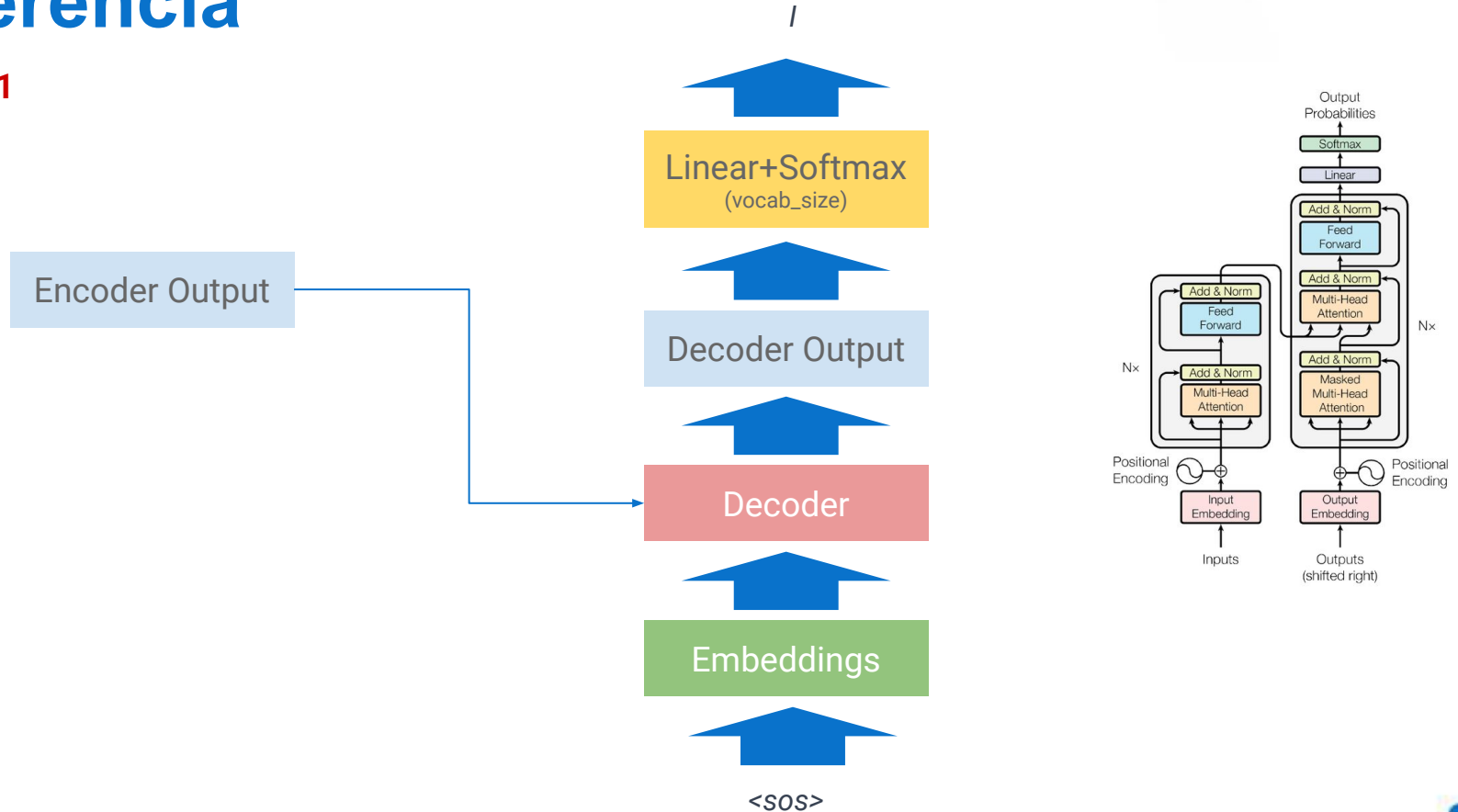


<sos> I like nachos



# Inferencia

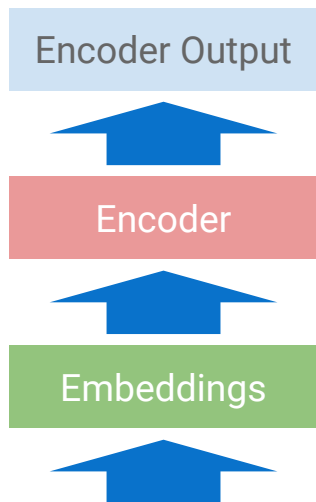
## PASO 1



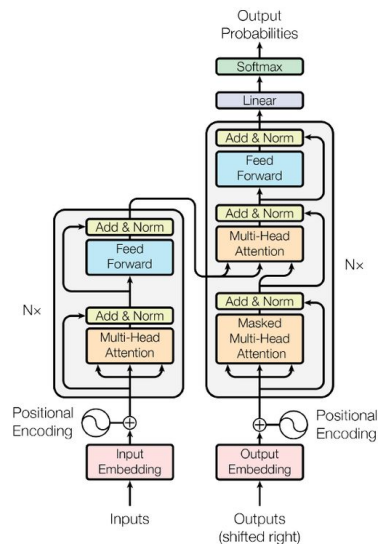
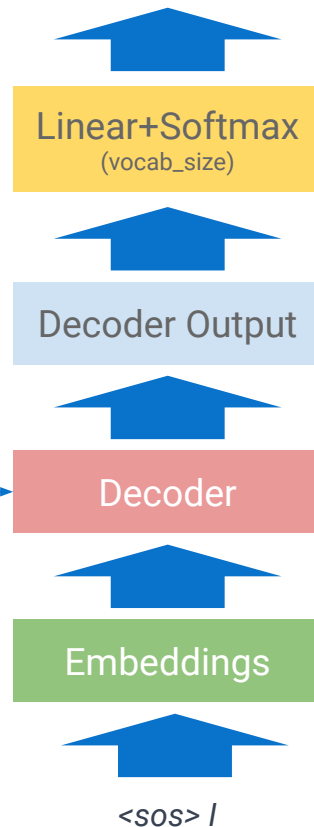
# Inferencia

## PASO 2

<sos> Me gustan los nachos <eos>

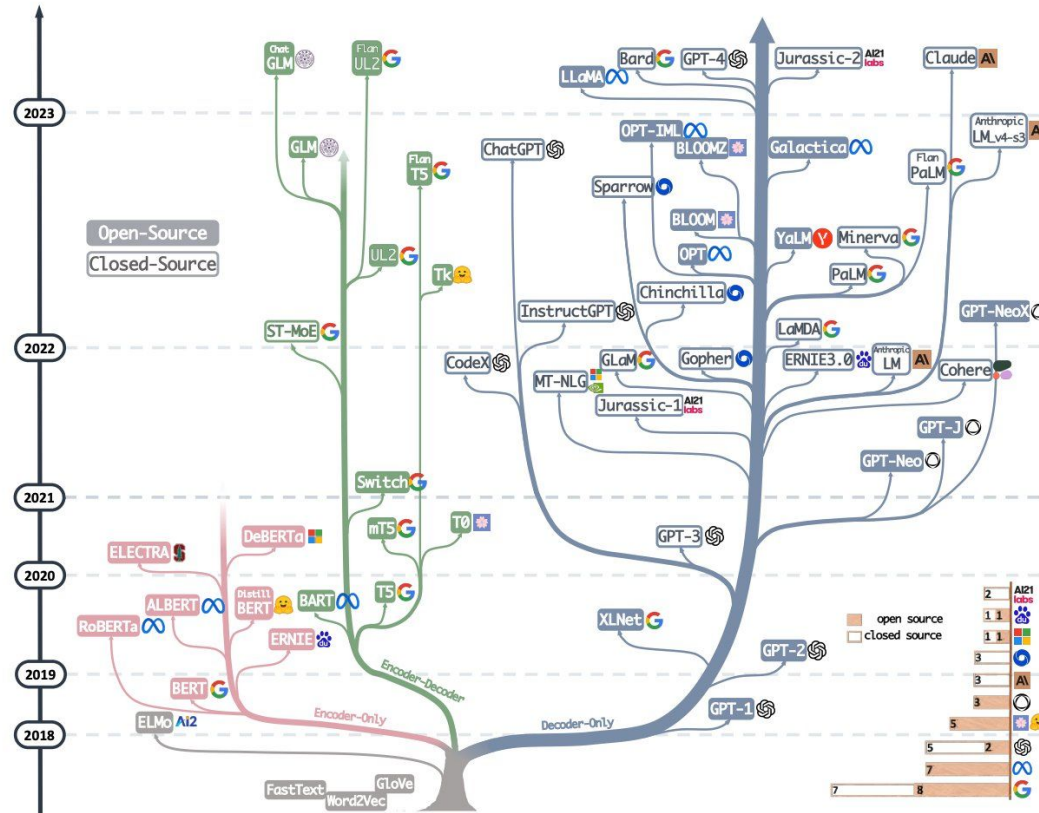


like



# Transformers timeline

(hasta 2023)





# Further Readings

Este tema acaba con la arquitectura de Transformers, que ha tenido un impacto muy grande en Deep Learning y Generative AI. Sin embargo, hay que tener en cuenta que cada modelo entrenado con esta arquitectura como base, tiene características únicas:

- No todos los modelos usan encoders, o decoders,
- Las capas pueden variar,
- La parametrización podría ser diferente,
- El entrenamiento se puede hacer con diversas técnicas,
- El dataset de entrenamiento es clave para conseguir buenos resultados,
- Etc.



# Further Readings

De todas las casuísticas posibles, se propone al estudiante que trabaje por su lado los siguientes conceptos:

- **Positional Encoding**, que representa una capa adicional de la arquitectura Transformers.
- **Masked Language Models**, que es una técnica de “enmascaramiento” de tokens para entrenar modelos como Bert y que consigue muy buenos resultados para entender el contexto.
- **Next Sentence Prediction**, que es otra técnica de entrenar modelos presentando pares de oraciones como entrada, y se le enseña a predecir si la segunda oración en el par es la continuación lógica de la primera.

Finalmente, con estos conocimientos, se propone al estudiante que analice en profundidad la arquitectura interna de los algoritmos: **Bert** y **GPT** (en este orden).



