

# Attention mechanisms & Transformers

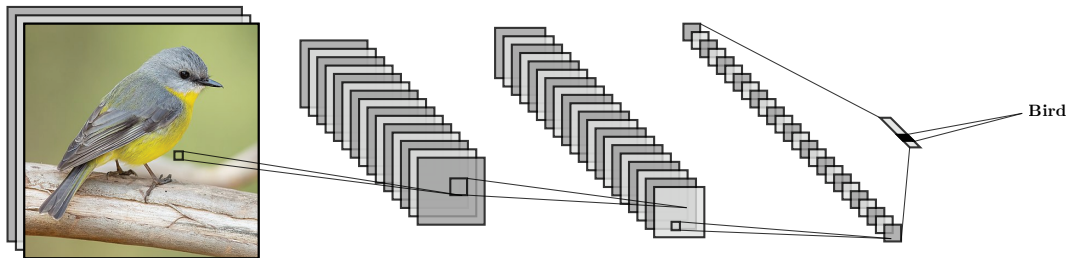
---

# Introducción

## Motivación

Muchas tareas no necesitan de toda la entrada para predecir la salida.

**Ejemplo:** Predecir la clase de una imagen.

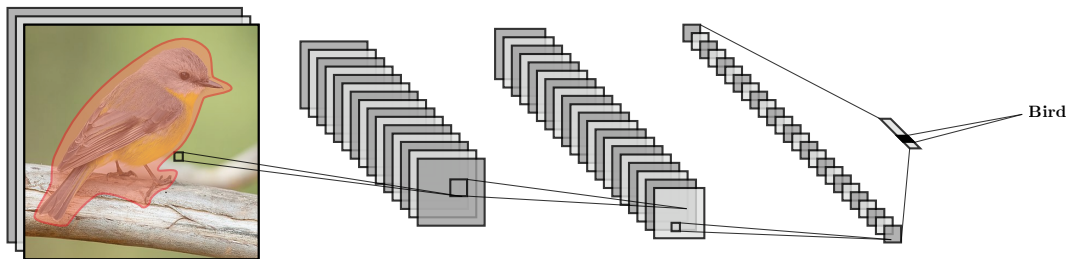


# Introducción

## Motivación

Muchas tareas no necesitan de toda la entrada para predecir la salida.

**Ejemplo:** Predecir la clase de una imagen.

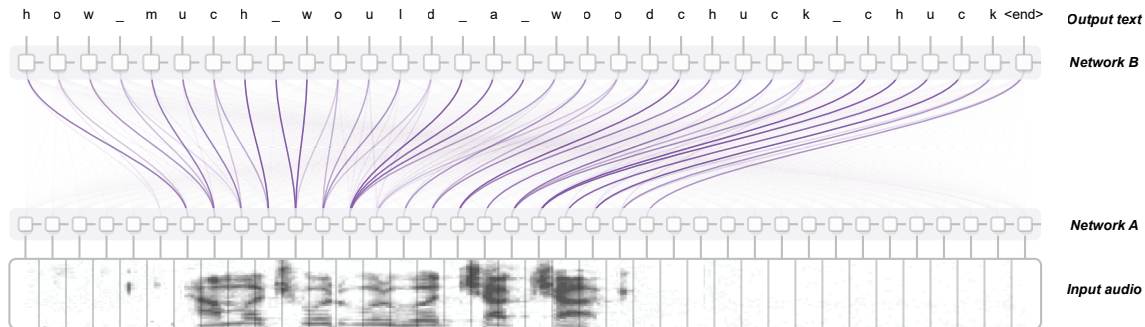


# Introducción

## Motivación

Muchas tareas no necesitan de toda la entrada para predecir la salida.

**Ejemplo:** Transformar audio en texto.

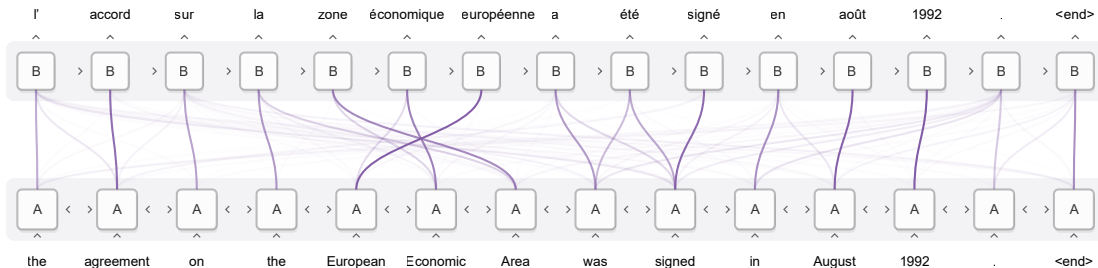


# Introducción

## Motivación

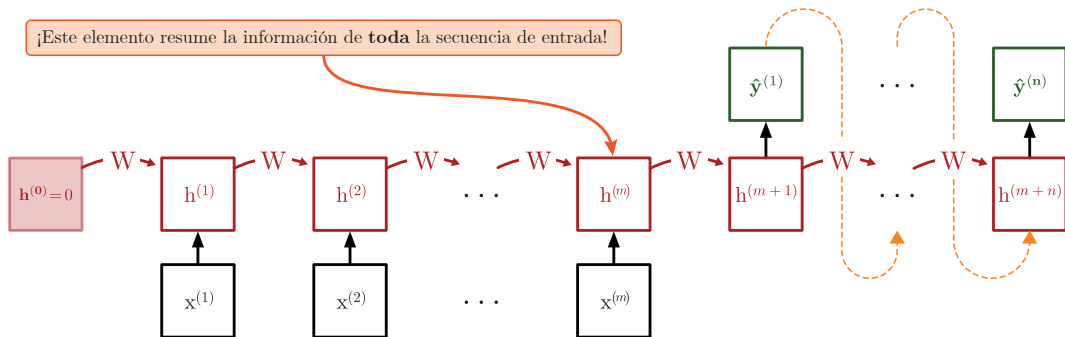
Muchas tareas no necesitan de toda la entrada para predecir la salida.

**Ejemplo:** Traducir entre idiomas.



## Motivación

En tareas de Secuencia a Secuencia, las RNN condensan toda la información de la entrada en un único elemento. No es la mejor opción, sobre todo en largas secuencias.



**En este contexto surgen los Transformers<sup>1</sup>.**

Esta nueva arquitectura:

- Mejora la eficiencia computacional de las RNN.
- Permite al modelo centrarse en partes concretas de la entrada para predecir la salida.
- Soluciona el problema de la memoria corto-placista de las RNN:
  - Permiten asociar palabras en una secuencia aunque estén muy separadas entre sí.

---

<sup>1</sup>Attention is all you need, Ashish Vaswani et al

Attention mechanisms & Transformers

---

# Attention mechanisms



# Attention mechanisms

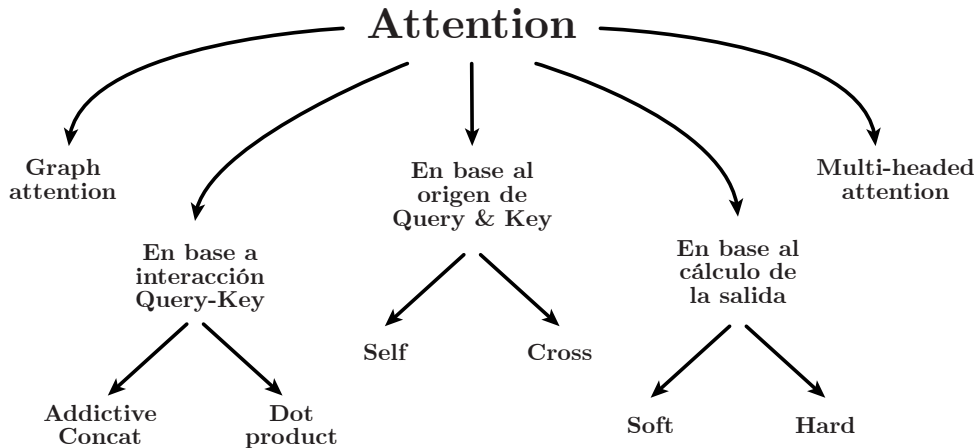
Antes de comenzar a hablar de *Transformers*, es necesario entender el funcionamiento de su componente principal, los **attention mechanisms**.

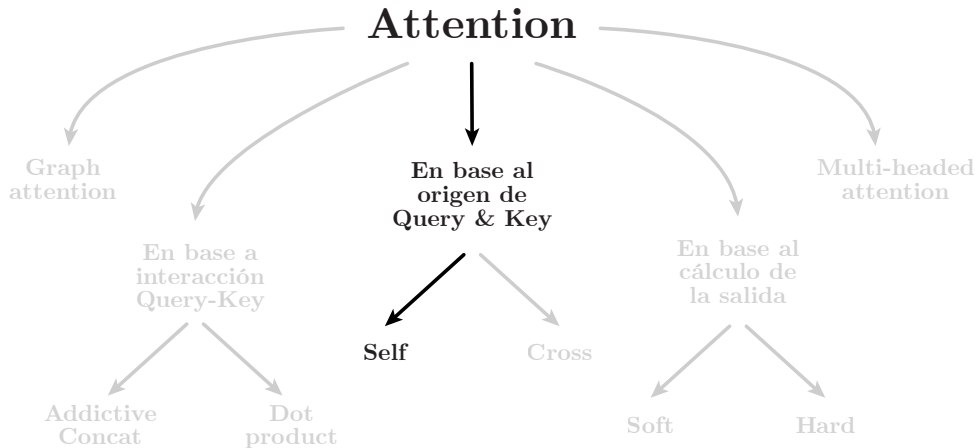
## Definición

Los mecanismos de atención seleccionan que elementos de la(s) secuencia(s) de entrada son más importantes para predecir la secuencia salida.

Detalles:

- La **entrada** de estos mecanismos espera **una o varias secuencias de datos**.
- Dentro de los *Transformers* se utilizan la llamada *Self-attention* pero, como verás a continuación, existen muchas otras variaciones.





# Self-attention

Este método requiere de los siguientes elementos:

- **Secuencia de entrada:**  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t$
- **Secuencia de salida:**  $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_t$
- Misma dimensión  $k$  para todos los vectores.

Para producir cada vector  $\mathbf{y}_i$  de la secuencia de salida, simplemente se obtiene la media ponderada de las entradas.

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j$$

Donde la  $j$  recorre toda la secuencia y la suma de todos los  $w_{ij}$  es igual a 1.

# Self-attention

El peso  $w_{i,j}$  **no es un parámetro**, como en una DNN, se deriva de una función sobre  $\mathbf{x}_i$  y  $\mathbf{x}_j$ .

La opción más sencilla para esta función es el **producto escalar**:

$$w'_{ij} = \langle \mathbf{x}_i^T, \mathbf{x}_j \rangle$$

El peso representa la importancia de cada elemento de la entrada para el elemento actual.

- Nótese que  $\mathbf{x}_i$  es el vector de entrada en la misma posición que el vector de salida actual.
- Para  $\mathbf{y}_{i+1}$ , obtenemos una serie completamente nueva de productos escalares y una suma ponderada diferente.

El producto escalar anterior nos da valores entre  $[-\infty, \infty]$ .

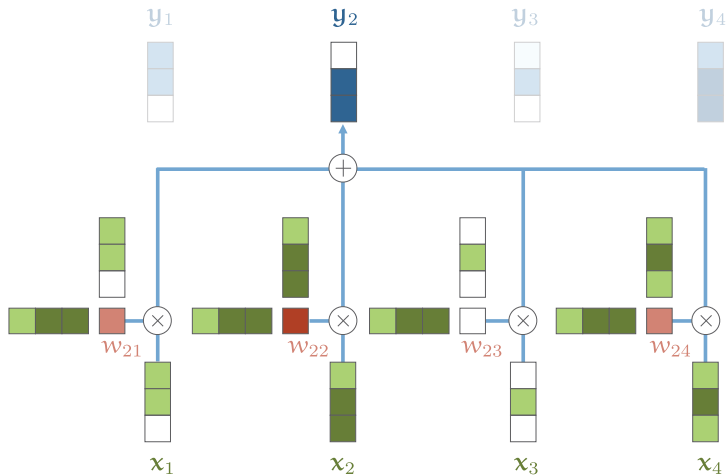
- Para obtener valores entre  $[0, 1]$ , aplicamos una *softmax*.
- De esta forma, para cada  $i$ , todos los  $j$  pesos sumarán 1.

Finalmente:

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}}$$

# Self-attention

De forma gráfica (softmax omitida por simplicidad):

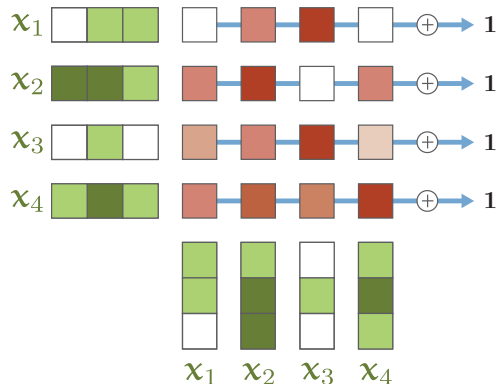


# Self-attention

Realizando este proceso para todos los  $x_i$  obtendremos una matriz de pesos como la representada en la figura.

Nótese que:

- Esta matriz se conoce como **matriz de atención**.
- Tras aplicar la softmax, todas las filas de esta matriz suman 1.
- A causa de esta softmax, la matriz no tiene por que ser simétrica.

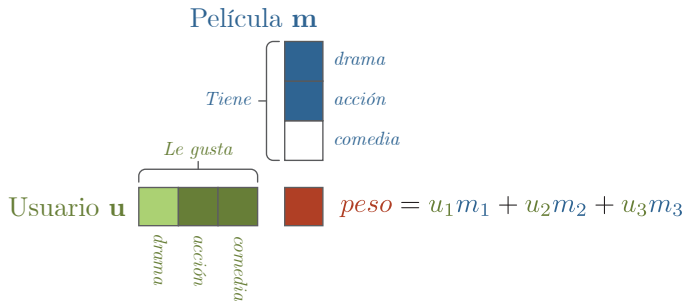




## ¿Por qué funciona la attention?

Supongamos que diriges un videoclub, tienes películas **m**, usuarios **u**, y te gustaría **recomendar** películas a tus usuarios que es probable que disfruten.

- Necesitamos codificar cada usuario y película de forma numérica.
- Podemos hacerlo de forma manual en base a los géneros.



# Ejemplo

## Importancia del signo:

Si **m** es romántica y a **u** le encanta el romanticismo o viceversa: *Producto escalar positivo*.

Si **u** es romántica y **u** odia el romanticismo o viceversa: *Producto escalar negativo*.

## Importancia de la magnitud:

Las magnitudes de los géneros indican cuánto contribuye a la puntuación total.

- Una película puede ser un poco romántica, pero no de forma notable.
- Un usuario puede no preferir el romanticismo, pero ser en gran medida ambivalente.

# Ejemplo

Rellenar manualmente estos valores es muy costoso y prácticamente imposible cuando existen millones de películas y usuarios.

## Para solucionarlo:

- 1 Las características de cada **m** y **u** pasarán a ser parámetros del modelo.
- 2 Pedimos a los usuarios que valoren varias películas.
- 3 Optimizamos los parámetros/características para que el producto escalar coincida con la valoración.

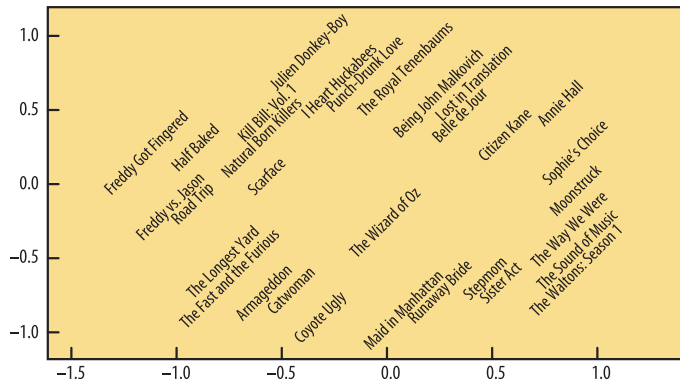
## Atención!

Las características de cada **u** y **m** ya no representan géneros, desconocemos su significado.

A pesar de ello, estas reflejan una semántica significativa sobre el contenido de la película.

# Ejemplo

Si representamos cada  $m$  con 2 de las 3 nuevas características aprendidas por el modelo:



**El modelo es capaz de juntar películas similares sin conocer nada sobre su contenido.**

# Self-attention

Este principio es el mismo que hace que la self-attention funcione.

Imaginemos que tenemos la secuencia de palabras (frase): “El gato camina en la calle”.

Para aplicar self-attention:

- 1 Representamos cada palabra por un vector  $\mathbf{v}$  (también llamado *embedding*) de tamaño  $k$ .

$$\mathbf{v}_{el}, \mathbf{v}_{gato}, \mathbf{v}_{camina}, \mathbf{v}_{en}, \mathbf{v}_{la}, \mathbf{v}_{calle}$$

- 2 Los valores de ese vector se aprenderán durante el entrenamiento (como ej. anterior).
- 3 Aplicamos self-attention a la secuencia, lo que retorna:

$$\mathbf{y}_{el}, \mathbf{y}_{gato}, \mathbf{y}_{camina}, \mathbf{y}_{en}, \mathbf{y}_{la}, \mathbf{y}_{calle}$$

donde  $\mathbf{y}_{gato}$  es la suma ponderada de todos los embeddings de la primera secuencia, ponderada por su producto escalar (normalizado) con  $\mathbf{v}_{gato}$ .

## Importante

Como estamos aprendiendo los valores de  $\mathbf{v}_t$ , el grado de “relación” entre dos palabras está **totalmente determinado por la tarea a resolver**.

Analizando la frase anterior, *en términos generales* podemos esperar que:

- El artículo “*El*” no sea muy relevante para el resto de palabras de la frase.
  - Su embedding  $\mathbf{v}_{El}$  tendrá un producto escalar bajo o negativo con todas las demás palabras.
- Para interpretar el significado de “*camina*” es muy útil averiguar quién está caminando.
  - Probablemente  $\mathbf{v}_{camina}$  y  $\mathbf{v}_{gato}$  tendrá un producto escalar alto y positivo.

## En resumen:

- Como se ve, el producto escalar expresa cómo de “relacionados” están dos vectores en la secuencia de entrada.
- El grado de “relación” viene **definido por la tarea de aprendizaje**.
- Los vectores de salida son **sumas ponderadas** sobre toda la secuencia de entrada.

## ¿Eso es todo?:

- **No hay parámetros que aprender (por ahora):** La parte de atención no aprende ningún parámetro. La codificación de la secuencia de entrada no forma parte del mecanismo.
- **La entrada es un conjunto, no una secuencia:** Si alteramos el orden de las palabras, la salida será la misma, solo que también permutada. Más adelante veremos como solucionarlo.

# Self-attention: Mejoras

La self-attention que se utiliza dentro de los Transformers utiliza **tres mejoras adicionales**.

- 1 Queries, keys y values.
- 2 Escalado del producto escalar.
- 3 Multi-head attention.

A continuación veremos cada una de ellas en detalle.



# Self-attention: Queries, keys y values

## Tres representaciones

Cada vector  $\mathbf{x}_i$  de la entrada se utiliza de tres formas diferentes dentro de la self-attention.

- **Query:** Se compara con otros vectores para establecer los pesos de su propia salida  $\mathbf{y}_i$ .
- **Key:** Se compara con otros vectores para establecer los pesos de la  $j$ -ésima salida  $\mathbf{y}_j$ .
- **Value:** Se usa en el cálculo de la media ponderada que retorna el vector de salida.

En los ejemplos que vimos hasta ahora, el vector  $\mathbf{x}_i$  *ejercía de todos estos roles a la vez*. Para facilitar la tarea a la atención, vamos a aprender **un embedding para cada rol**.

## Self-attention: Queries, keys y values

Para aprender estas representaciones aplicaremos una transformación lineal al vector original.

Crearemos tres matrices de tamaño  $k \times k$ :  $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ .

Ahora, para cada elemento  $\mathbf{x}_i$  de la secuencia de entrada tendremos *tres embeddings*:

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \quad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \quad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

**¿Dónde utilizarlos en self-attention?**

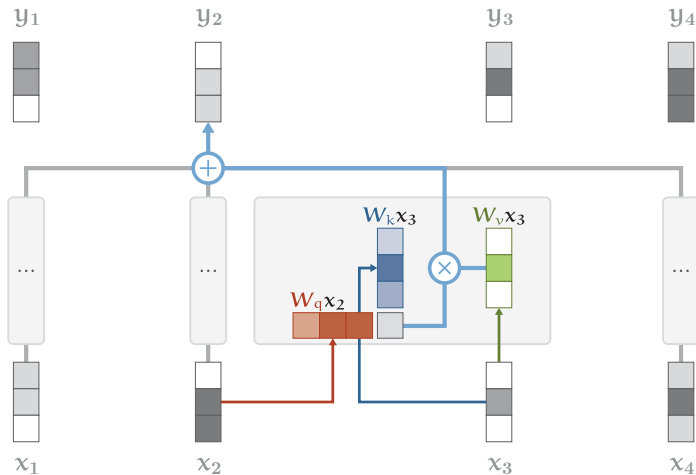
$$w'_{ij} = \langle \mathbf{q}_i^T, \mathbf{k}_j \rangle \quad w_{ij} = \text{softmax}(w'_{ij}) \quad \mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j$$

El producto escalar se hace entre *query* y *key*, para la media ponderada se utilizan los *values*.

**Estas tres matrices serán los parámetros que aprende la self-attention.**

# Self-attention: Queries, keys y values

De forma gráfica:



# Self-attention: Escalado del producto escalar

# Self-attention: Multi-head

# Cross-attention

Attention mechanisms & Transformers

---

# Transformers

