

# 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.



# Introducción

## 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.



# Introducción

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.

# Variaciones



# 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.

## Self-attention

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

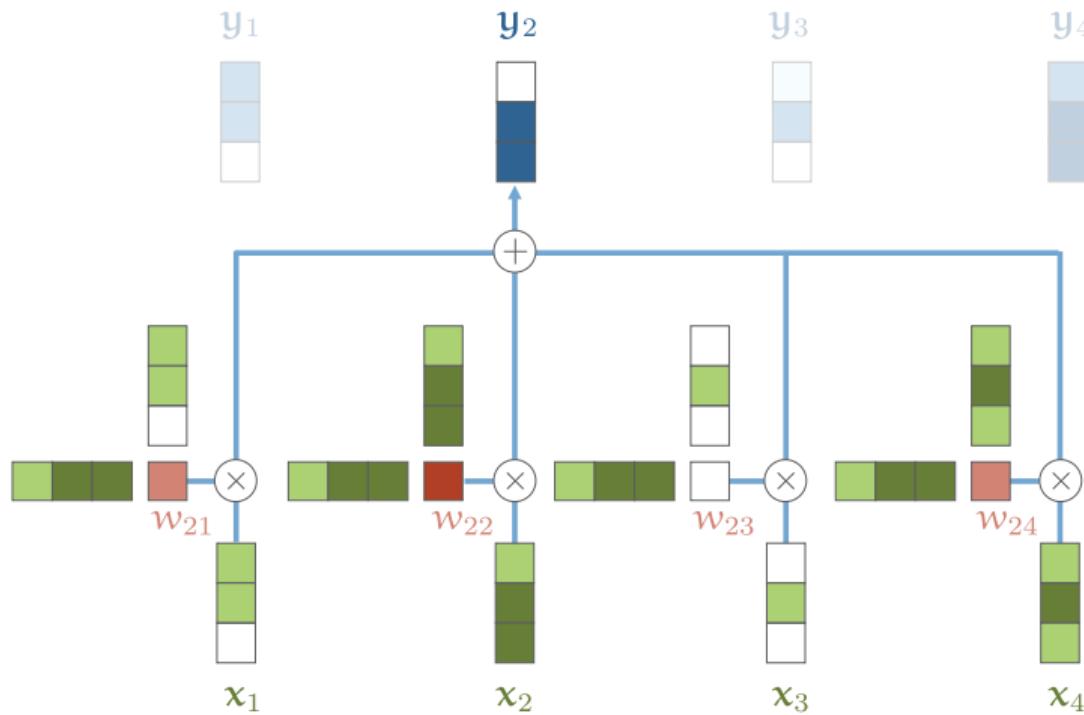
- Para obtener valores entre  $[0, 1]$ , aplicamos una *softmax*.
- De esta forma, para cada  $i$ , todos los  $j$  pesos sumarán 1.

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 qué ser simétrica.



# Ejemplo

## ¿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  $v_t$ , 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  $v_{El}$  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  $v_{camina}$  y  $v_{gato}$  tendrá un producto escalar alto y positivo.

# Self-attention

## 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  $x_i$  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  $y_i$ .
- **Key:** Se compara con otros vectores para establecer los pesos de la  $j$ -ésima salida  $y_j$ .
- **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  $x_i$  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?

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

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

## Problema del softmax

La función softmax puede ser sensible a valores de entrada muy grandes. Estos perjudican el gradiente y ralentizan o detienen el aprendizaje.

- Aumentar el tamaño  $k$  de los embeddings, aumenta el valor medio del producto escalar.
- Hay que reducir este valor escalando el resultado:

$$w'_{ij} = \frac{\langle \mathbf{q}_i^T, \mathbf{k}_j \rangle}{\sqrt{k}}$$

## ¿Por qué $\sqrt{k}$ ?

- Dividir por la raíz cuadrada del tamaño del embedding normaliza los valores.
- Normalizar escala los valores evitando que unos dominen o se anulen otros.

# Self-attention: Multi-head



# Self-attention: Multi-head

## Problema

Una palabra puede significar cosas distintas para vecinos distintos.

**Ejemplo:** “*Marta da rosas a Sara*”.

- $x_{Marta}$  y  $x_{Sara}$  influirán en  $x_{da}$  en diferente cantidad, pero no de *diferente forma*.
- Si queremos que la información sobre quien dio las rosas y quien las recibió acabe en diferentes partes de  $x_{da}$  necesitamos *más flexibilidad*.

## Solución:

- Combinar  $r$  mecanismos de self-attention para mejorar la capacidad de discriminar.
- Por tanto se aprenderán  $r$  matrices *query*, *key* y *value*:  $\mathbf{W}_q^r$ ,  $\mathbf{W}_k^r$ ,  $\mathbf{W}_v^r$ .

Cada uno de estos mecanismos se denomina “cabeza” o “head”.

# Self-attention: Multi-head

## Múltiples heads, múltiples salidas

Cada  $x_i$  produce un vector de salida  $y_i^r$  diferente en cada self-attention head.

Para obtener una salida del mismo tamaño que la entrada:

- 1 Concatenamos las  $i$ -ésimas salidas de cada head  $\forall r$ .
  - Esto nos dará un vector de  $r \times k$  elementos para cada entrada.
- 2 Transformamos linealmente de nuevo a tamaño  $k$ .

Necesitaremos, por tanto, una nueva matriz de pesos que denominaremos  $W_o$ .

# Self-attention: Multi-head

La multi-head attention se puede ver como  $r$  copias de self-attention aplicadas en paralelo.

## Problema:

- Cada copia tiene su propia *query*, *key* y *value*.
- Mejor rendimiento, pero  $r$  veces más lento que una sola cabeza.

Si el vector  $\mathbf{x}_i$  tiene dimensión  $k = 256$  y tenemos  $r = 4$  heads:

Entrada	Proyección	q,k,v	Heads	Parámetros
256	256	3	1	196608
256	256	3	4	786432

# Self-attention: Multi-head eficiente

## Solución:

- Reducir la dimensión de las proyecciones *query*, *key* y *value*.
- Transformamos cada  $x_i$  a tamaño 64 ( $256/4$ ) para cada *query*, *key* y *value*.

Entrada	Proyección	q,k,v	Heads	Parámetros
256	256	3	1	196608
256	256	3	4	786432
256	64	3	4	196608

Ojo, para estos ejemplos estamos omitiendo la matriz  $\mathbf{W}_o$ .

# Self-attention: Multi-head

Finalmente, todo el proceso se puede representar como:



# Self-attention: Multi-head

Finalmente, todo el proceso se puede representar como:



# Self-attention: Multi-head

Finalmente, todo el proceso se puede representar como:



# Cross-attention



# Self vs Cross-attention

Como se ha visto, la self-attention busca:

- La importancia de cada elemento de la secuencia para el resto de elementos de la misma.

¿Y si el problema requiere mezclar secuencias?

En algún caso puede ser interesante obtener la importancia de cada elemento **de una secuencia** para **otra secuencia** diferente. Esto se conoce como **cross-attention**.

## Cambios:

- Dos secuencias de entrada: Una con elementos  $x_i$  y otra con  $z_i$ .
- Los *query* se obtienen de la segunda secuencia.
- Los *key* y *value* de la primera.

# Cross-attention

De forma gráfica:



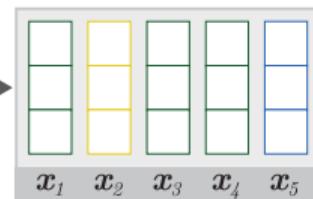
# Cross-attention

## Aplicaciones típicas:

- **Transformers:** Se suele ubicar en el decoder.
- **Visual Question Answering (VQA):** Responder preguntas sobre imágenes:



¿De qué color es  
el balón de fútbol?



*Cross-attention layer*

Attention mechanisms & Transformers

---

## Transformers

# Transformers

Una vez conocido el funcionamiento de los **mecanismos de atención**, podemos empezar a hablar del **Transformer**.

Presentado en 2017 por Vaswani et al. Fué diseñado para tareas de traducción (seq-2-seq), pero hoy en día es clave en la mayoría de tareas NLP.

## El Transformer:

- Es una arquitectura (conjunto de capas).
- Utiliza mecanismos de self y cross-attention.
- Es altamente paralelizable: permite entrenar modelos más grandes en menos tiempo.

# Transformers

El Transformer tradicional se compone de dos módulos:

- Encoder
- Decoder

Cada uno está formado por:

- Capas self-attention.
- Capas cross-attention.
- Capas de normalización.
- Perceptrones multicapa.
- Conexiones residuales.



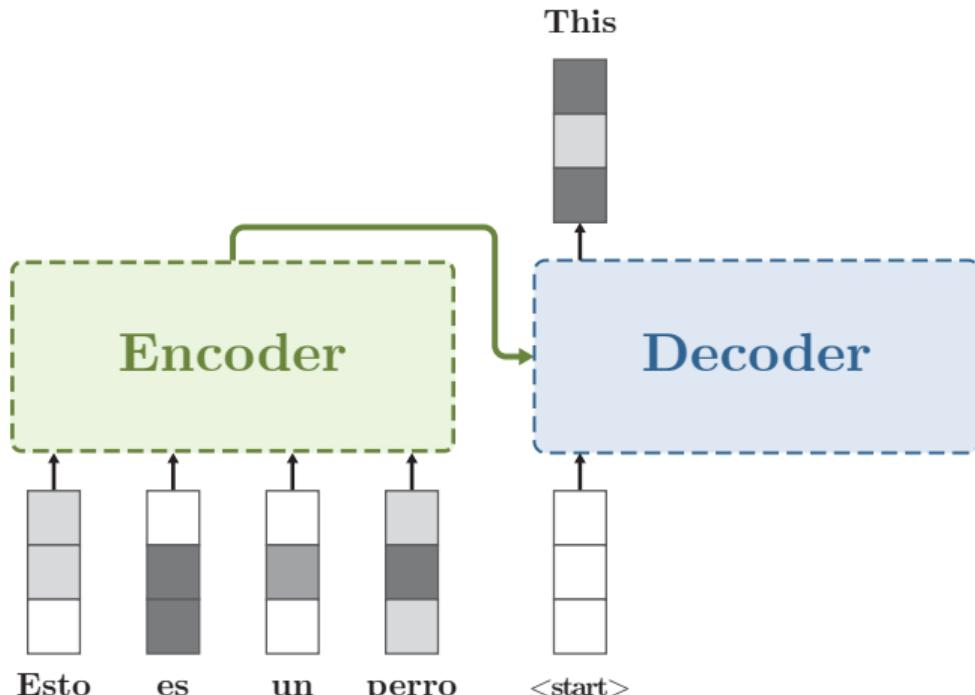
# Transformers

## El encoder:

- Recibe secuencia de entrada.
- Genera secuencia codificada.

## El decoder:

- Recibe salida del encoder.
- Recibe su salida anterior.
  - Primero recibe *< start >*.
- Genera hasta *< end >*.



# Transformers

## El encoder:

- Recibe secuencia de entrada.
- Genera secuencia codificada.

## El decoder:

- Recibe salida del encoder.
- Recibe su salida anterior.
  - Primero recibe *< start >*.
- Genera hasta *< end >*.



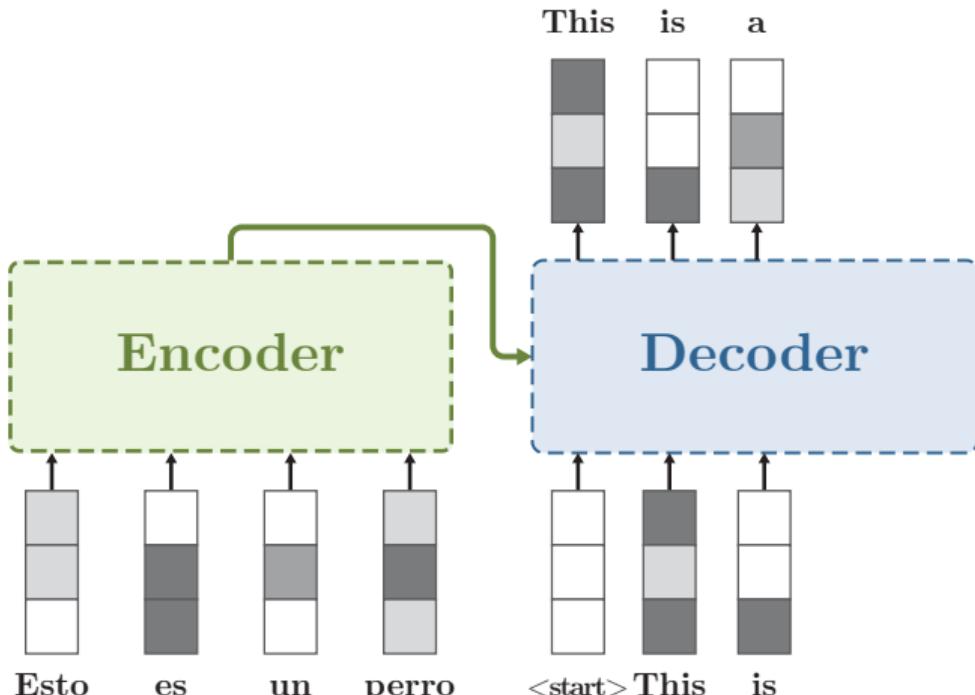
# Transformers

## El encoder:

- Recibe secuencia de entrada.
- Genera secuencia codificada.

## El decoder:

- Recibe salida del encoder.
- Recibe su salida anterior.
  - Primero recibe *< start >*.
- Genera hasta *< end >*.



# Transformers

## El encoder:

- Recibe secuencia de entrada.
- Genera secuencia codificada.

## El decoder:

- Recibe salida del encoder.
- Recibe su salida anterior.
  - Primero recibe *< start >*.
- Genera hasta *< end >*.



# Transformers

## El encoder:

- Recibe secuencia de entrada.
- Genera secuencia codificada.

## El decoder:

- Recibe salida del encoder.
- Recibe su salida anterior.
  - Primero recibe *< start >*.
- Genera hasta *< end >*.



# Transformers: Encoder

El **encoder** de un Transformer se compone **6 módulos** concatenados.

**Cada módulo contiene, por orden:**

- ① Capa multi-head self-attention.
- ② Conexión residual con entrada previa.
- ③ Capa de normalización.
- ④ Un MLP por cada elemento.
- ⑤ Conexión residual con entrada previa.
- ⑥ Capa de normalización.

# Transformers: Encoder

En detalle:



# Transformers: Decoder

El **decoder** de un Transformer se compone **6 módulos** concatenados.

**Cada módulo contiene, por orden:**

- ① Capa multi-head self-attention.
- ② Conexión residual con entrada previa.
- ③ Capa de normalización.
- ④ Capa multi-head **cross-attention (con salida de encoder)**.
- ⑤ Conexión residual con entrada previa.
- ⑥ Capa de normalización.
- ⑦ Un MLP por cada elemento.
- ⑧ Conexión residual con entrada previa.
- ⑨ Capa de normalización.

# Transformers: Decoder

En detalle:



# Transformers vs RNN

**RNN:** Como ya habíamos visto, en tareas seq2seq condensan toda la información de la entrada en un único elemento. Esto se traduce en una **memoria corto-placista**.



**Transformers:** Aprovechan toda la secuencia de entrada para generar la salida.

# Transformers vs RNN

**RNN:** Al depender un elemento del anterior, **no es posible paralelizar** su ejecución. Esto implica un menor rendimiento.



**Transformers:** Altamente paralelizables. Grandes modelos entrenados en menos tiempo.

# Transformers vs RNN

RNN: Tienen en cuenta el orden de los elementos de la secuencia de entrada.

Transformers:

- Tal como los hemos descrito hasta este punto, no tienen en cuenta el orden.
- Esto es una desventaja, el orden de los elementos es muy importante.
- Necesitamos incorporar algo más: **Positional encoding**.

# Transformers: Positional encoding

## Positional encoding:

- Sumar un vector  $\mathbf{t}_i$  a cada elemento  $\mathbf{x}_i$  de la entrada.
- Este vector ha de ser del mismo tamaño  $k$  que  $\mathbf{x}_i$ .

Los valores se obtienen:

$$\mathbf{t}_{i,2j} = \sin\left(\frac{i}{10000^{\frac{2j}{k}}}\right)$$

$$\mathbf{t}_{i,2j+1} = \cos\left(\frac{i}{10000^{\frac{2j}{k}}}\right).$$

Como se puede ver, los elementos pares del vector se obtienen de forma diferente a los impares.

# Transformers: Positional encoding

Visualización de los vectores  $\mathbf{t}_i$  en secuencias de tamaño máximo 50 con  $k = 128$ :



# Transformers: Positional encoding

Localización dentro del Transformer:



# Transformers: Aplicaciones

Diseñados inicialmente para resolver problemas de traducción. En la actualidad se utilizan en muchas más tareas.

## Natural Language Processing:

- Traducción
- Clasificación
- Conversación (chat)

## Imágenes:

- Detección de objetos
- Super-resolución

## Multimodal:

- Análisis de sentimientos (texto e imágenes)
- Vídeo y texto (por ejemplo, subtítulos automáticos para videos)
- Generación de descripciones para imágenes (Image captioning)

# Transformers: Aplicaciones

En tareas NLP, gracias a la introducción de los Transformers los modelos de lenguaje han crecido en tamaño, precisión y complejidad.



El modelo ELMo no utiliza transformers, está basado en LSTM bidireccionales.

# Transformers: Aplicaciones

## GPT-3 (Generative Pre-trained Transformer 3):

- Creado por OpenAI.
- Modelo de lenguaje con 175 mil millones de parámetros.
- Utilizado en generación de texto coherente, traducción y chatbots avanzados.



# Transformers: Aplicaciones

## CLIP (Contrastive Language-Image Pretraining):

- Modelo creado por OpenAI que combina imágenes y texto utilizando transformers.
- Capacidad para entender contenido visual y textual, útil en clasificación de imágenes y búsqueda basada en texto.



# Transformers: Aplicaciones

## BERT (Bidirectional Encoder Representations from Transformers):

- Creado por Google.
- Es bidireccional, teniendo en cuenta el contexto anterior y posterior de una palabra.
- Utilizado en tareas de procesamiento de lenguaje natural como clasificación, etiquetado de entidades y resolución de preguntas.



# Transformers: Aplicaciones

## T5 (Text-to-Text Transfer Transformer):

- Modelo de lenguaje de Google que sigue el enfoque “texto a texto”.
- Capaz de realizar múltiples tareas de procesamiento de lenguaje natural, como traducción, resumen y generación de respuestas.
- Versátil y eficiente al unificar diferentes tareas bajo una sola arquitectura.



# Referencias

- ① **Attention? An Other Perspective!, Nikhil Shah**
- ② **Lecture 7: Attention and transformers, Prof. Gilles Louppe**
- ③ **Transformers from scratch, Peter Bloem**
- ④ **The Illustrated Transformer**