

# Tema 6.3

## Redes con mecanismo de atención

Miguel Ángel Martínez del Amor

Deep Learning

Departamento de Ciencias de la Computación e Inteligencia Artificial

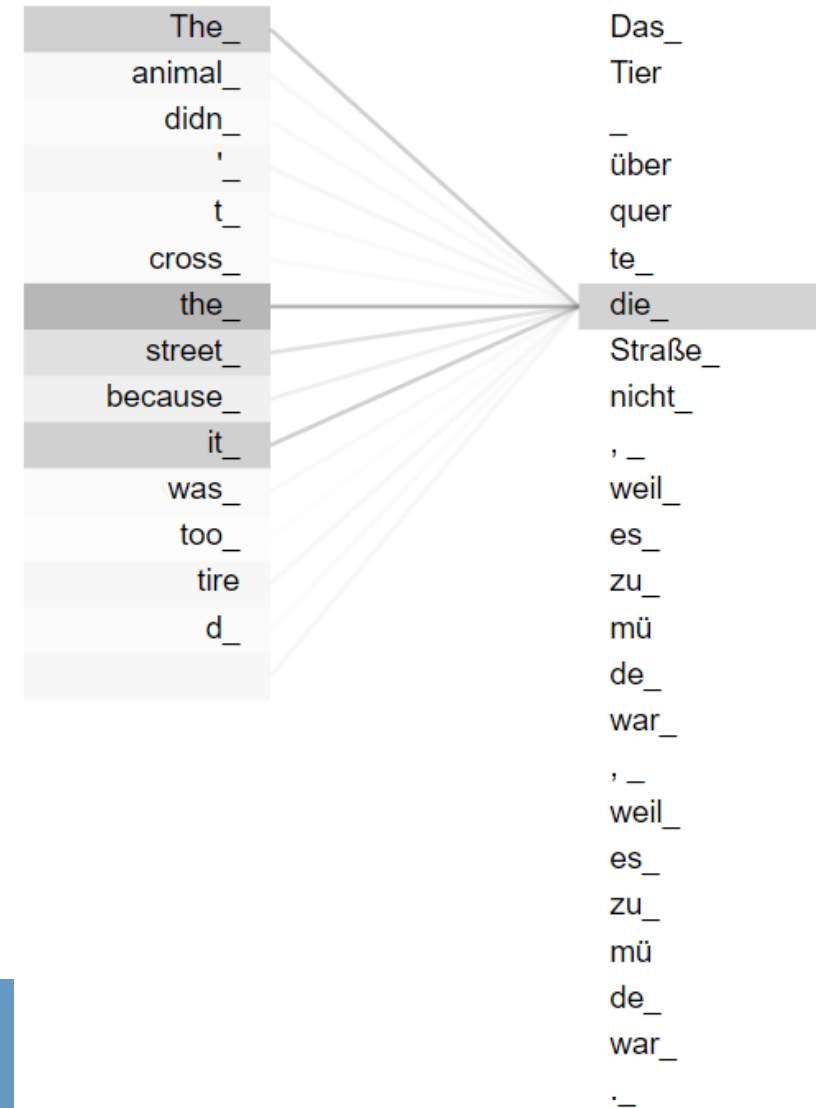
Universidad de Sevilla

# Contenido

- Idea intuitiva
- Seq2seq
- Attention en seq2seq
- Transformers

# Idea: attention vs redes recurrentes

- Supongamos por ejemplo la traducción de lenguaje (de español a inglés).
- **Redes recurrentes:** memorizar elementos de la secuencia anteriores para las próximas, usando distintas vías (forget gate, etc..)
- **Redes con attention** (transformers): siempre ven toda la secuencia de entrada y para cada elemento de salida, aprenden a ver qué elementos de entrada son importantes.

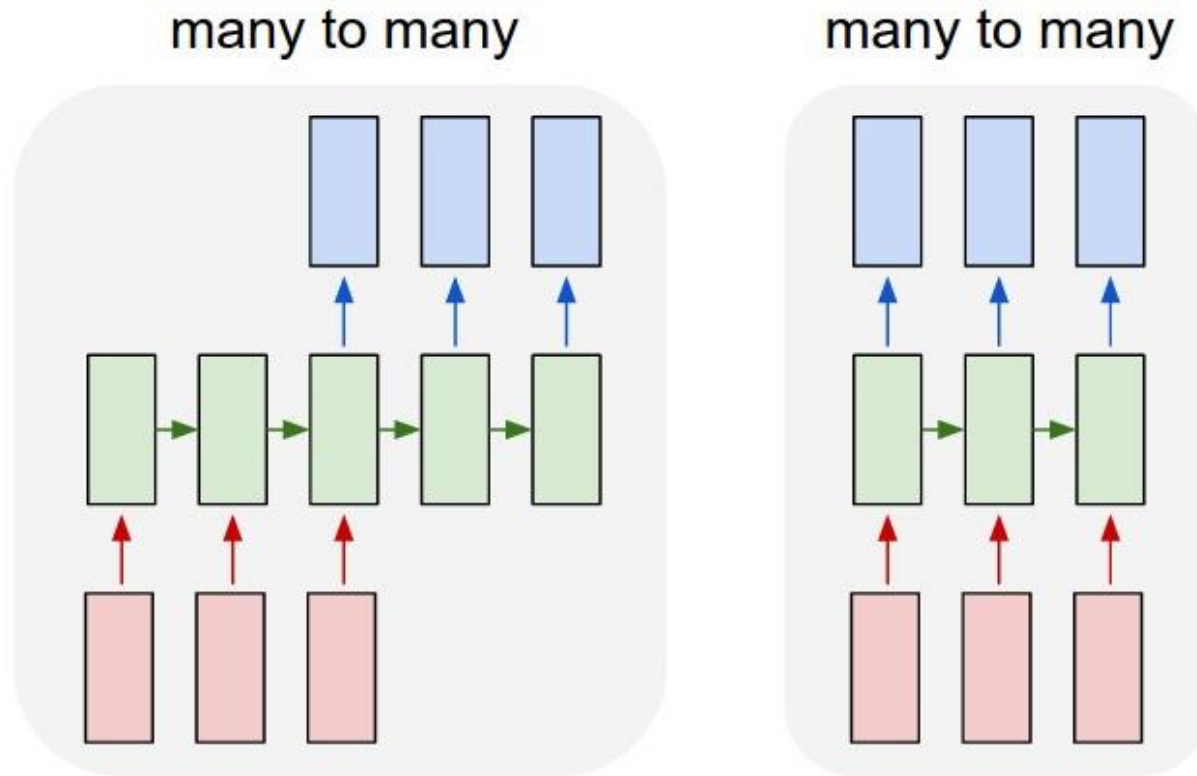


# Idea: attention vs redes recurrentes

- Las redes con **attention** surgen con el fin de solucionar ciertos problemas con las **LSTM**:
  - Computación **secuencial** que inhibe la paralelización entre los elementos de una secuencia de entrada.
  - **No** hay modelización **explícita** de **dependencias** a largo y corto plazo.
  - La **distancia** entre las posiciones de los **elementos** en la secuencia es lineal.

# Modelo Sequence to Sequence (seq2seq)

- Ya vimos la idea en un tema anterior:



# Modelo Sequence to Sequence (seq2seq)

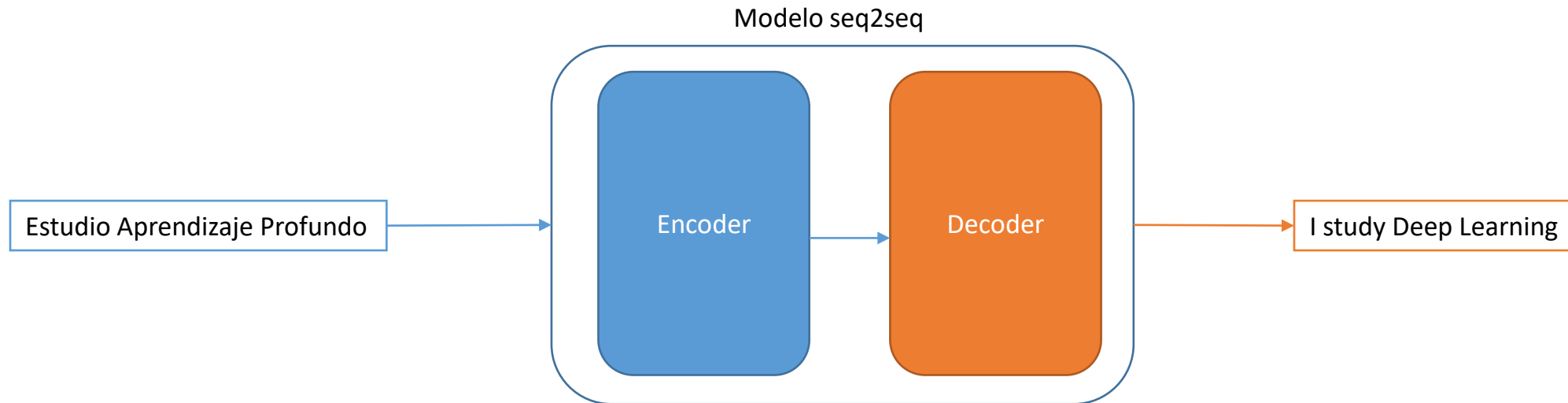
- Este ejemplo está basado en [este enlace](#)
- Un ejemplo en traducción de lenguaje:



([Sutskever et al., 2014](#), [Cho et al., 2014](#)).

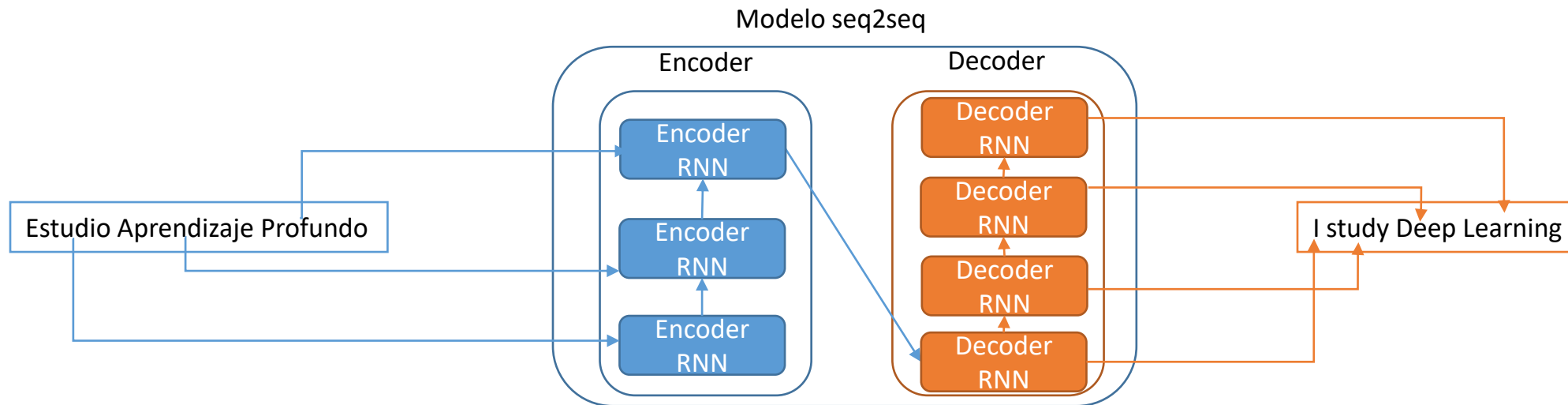
# Modelo Sequence to Sequence (seq2seq)

- Viendo por dentro el modelo, hay dos fases:
  - **Encoder** (*“el que entiende del lenguaje de entrada”*)
  - **Decoder** (*“el que entiende del lenguaje de salida”*)



# Modelo Sequence to Sequence (seq2seq)

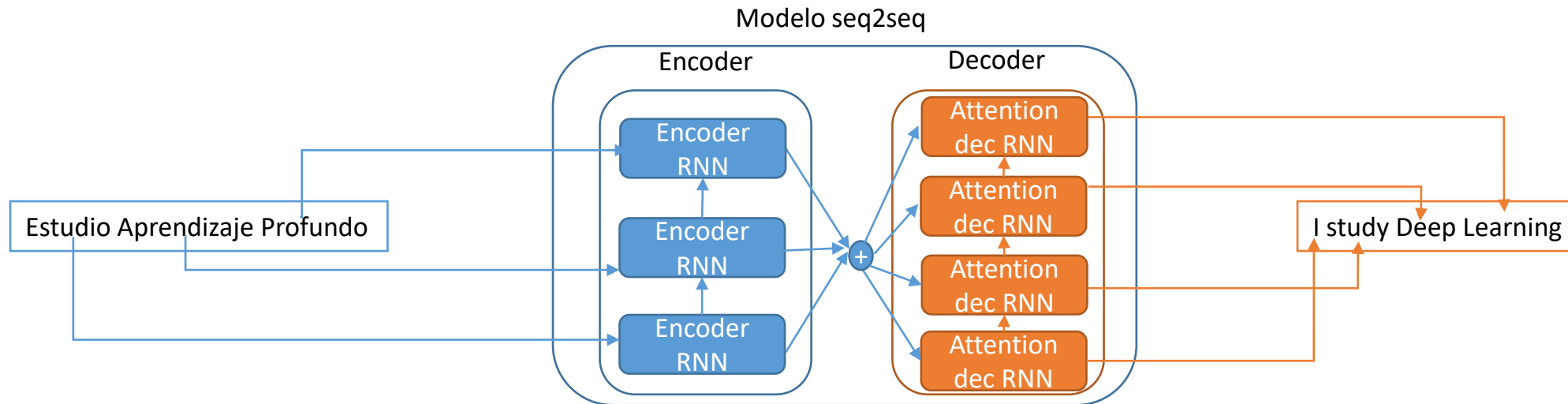
- Se pueden implementar con RNN (con unidades LSTM por ejemplo)
- El último **estado oculto** del **encoder** se pasa al **decoder**





# Attention en seq2seq

- Modelo con **attention**: todos los estados ocultos del encoder se pasan al decoder, ya que cada estado oculto del encoder está más asociado a cada palabra de entrada.

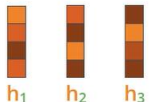


# Attention en seq2seq

- Idea intuitiva detrás del Attention:
  - En vez de codificar toda la frase en un solo estado oculto (como harían las RNN), **cada palabra** tiene un **estado oculto** que se pasa al decoder.
- La suma de los estados ocultos del encoder en realidad es una **suma ponderada** de cada una
  - Esto se hace para cada unidad de attention en el decoder

# Attention en seq2seq

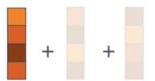

## Attention at time step 4

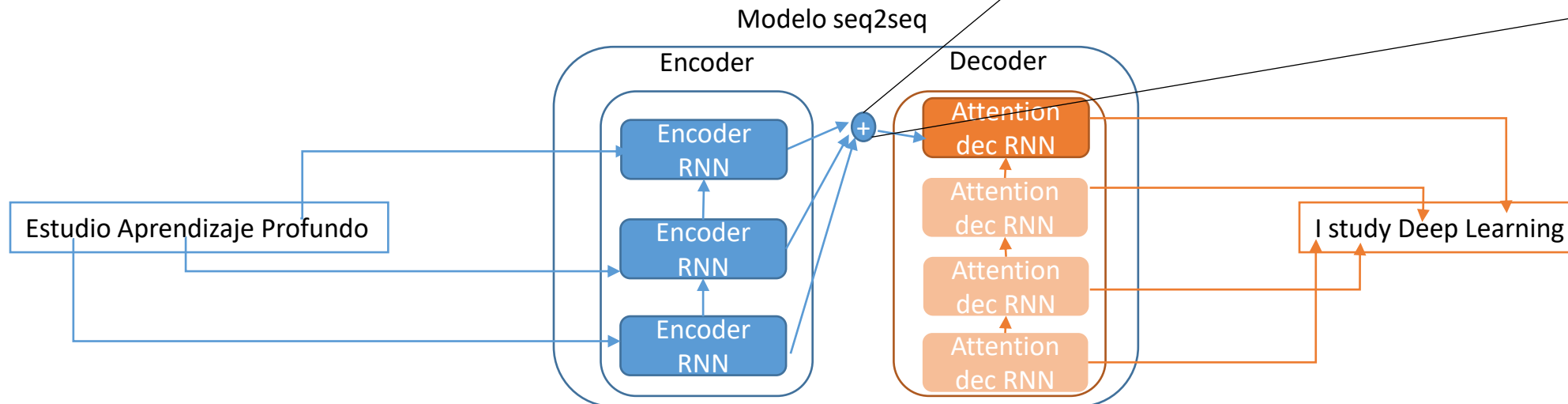
1. Prepare inputs  
  
 $h_1$   $h_2$   $h_3$  Encoder hidden states
2. Score each hidden state  

13	9	9
----	---	---

 scores  
Attention weights for decoder time step #4
3. Softmax the scores  

0.96	0.02	0.02
------	------	------

 softmax scores
4. Multiply each vector by its softmaxed score  
  
+ + +  
=  
  
Context vector for decoder time step #4
5. Sum up the weighted vectors



# Transformers

- Presentado en el famoso artículo: [“Attention is All You Need”](#) 2017.
- **Objetivos:**
  - Paliar la falta de paralelismo en las RNN sobre los elementos de la secuencia
  - No asumir que los elementos de la secuencia son equidistantes

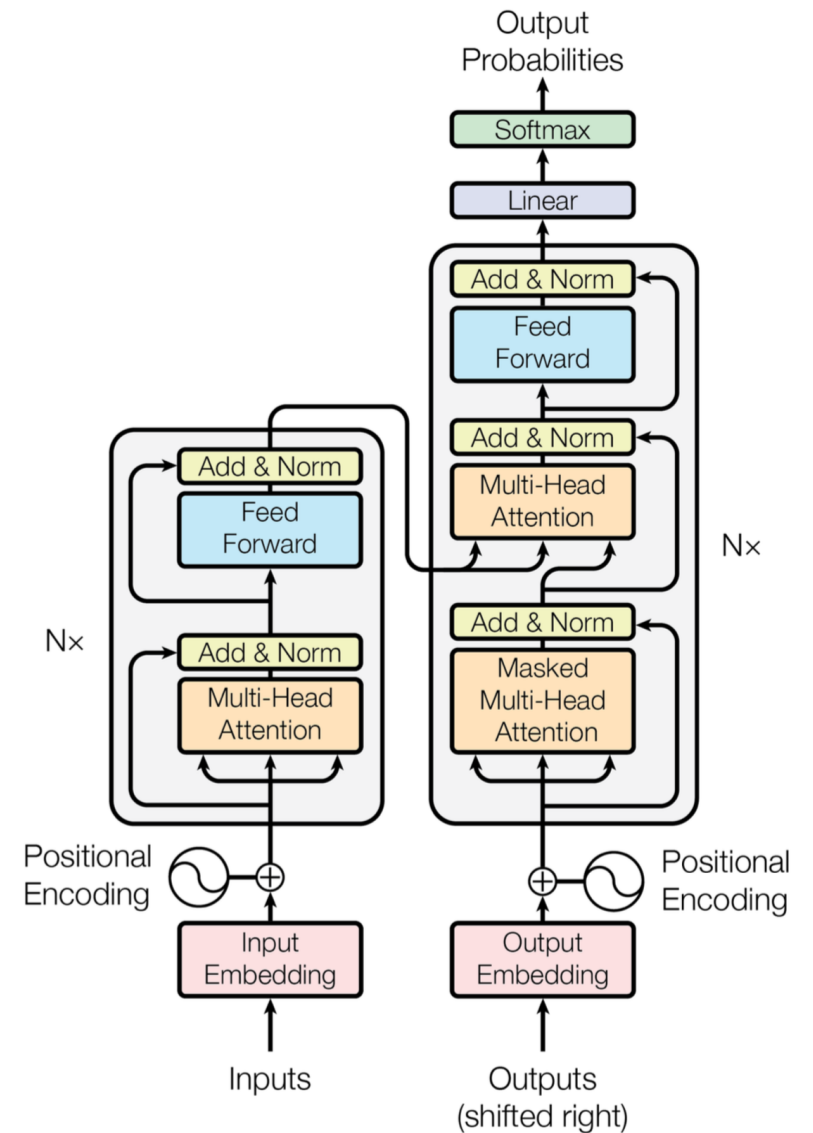


Figure 1: The Transformer - model architecture.

# Transformers

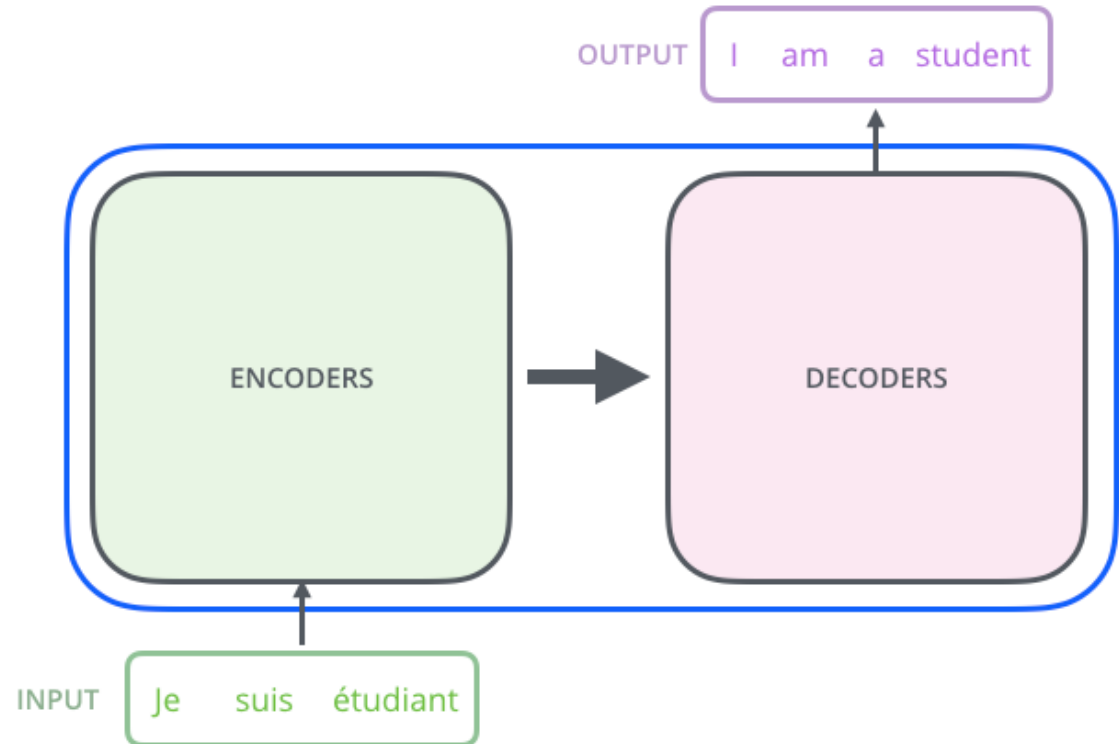
- Un **Transformer** recibe una secuencia como entrada y devuelve otra secuencia:
  - Veamos de nuevo como ejemplo la traducción de texto, pero podría ser cualquier otro tipo de secuencia (series temporales, píxeles de una imagen, etc)



<http://jalammar.github.io/illustrated-transformer/>

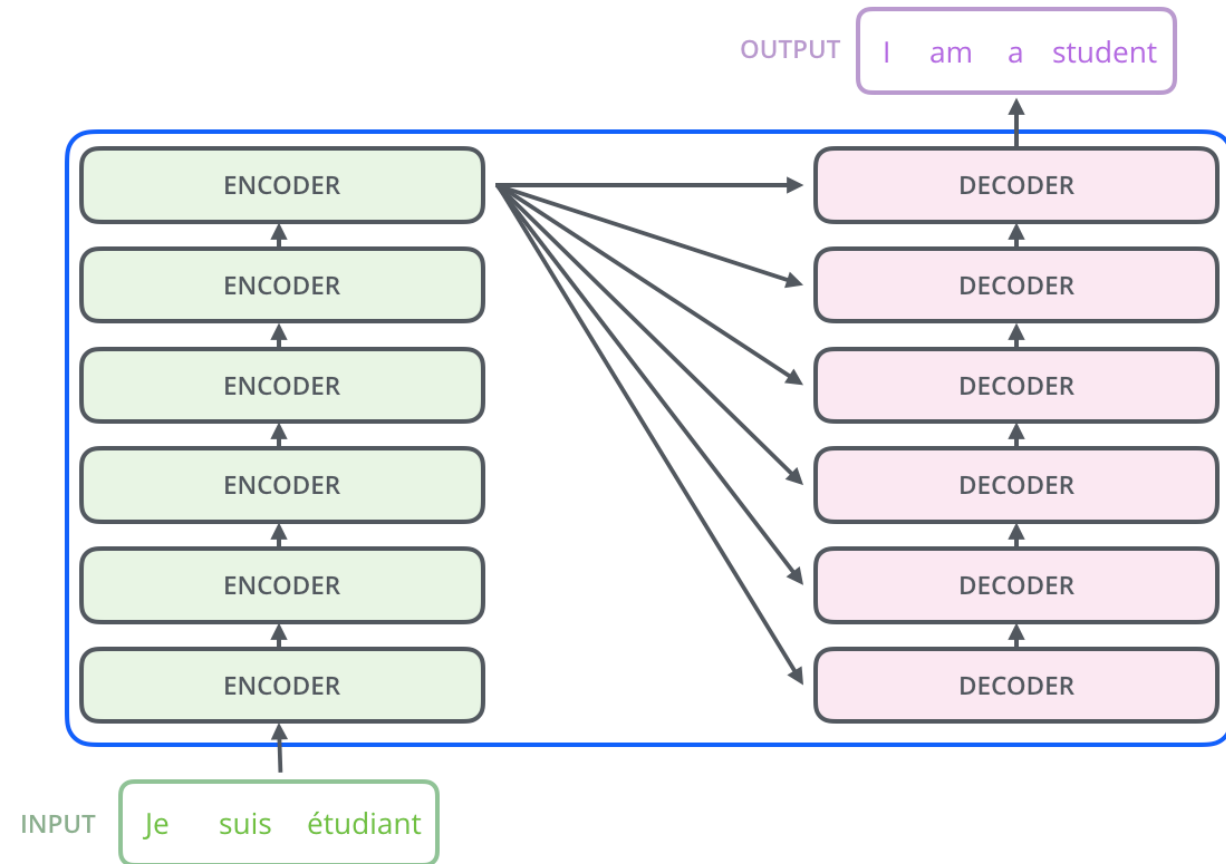
# Transformers

- Como en seq2seq, existe un **encoder** y un **decoder**:
  - El encoder recibe la secuencia de entrada
  - El decoder genera la secuencia de salida



# Transformers

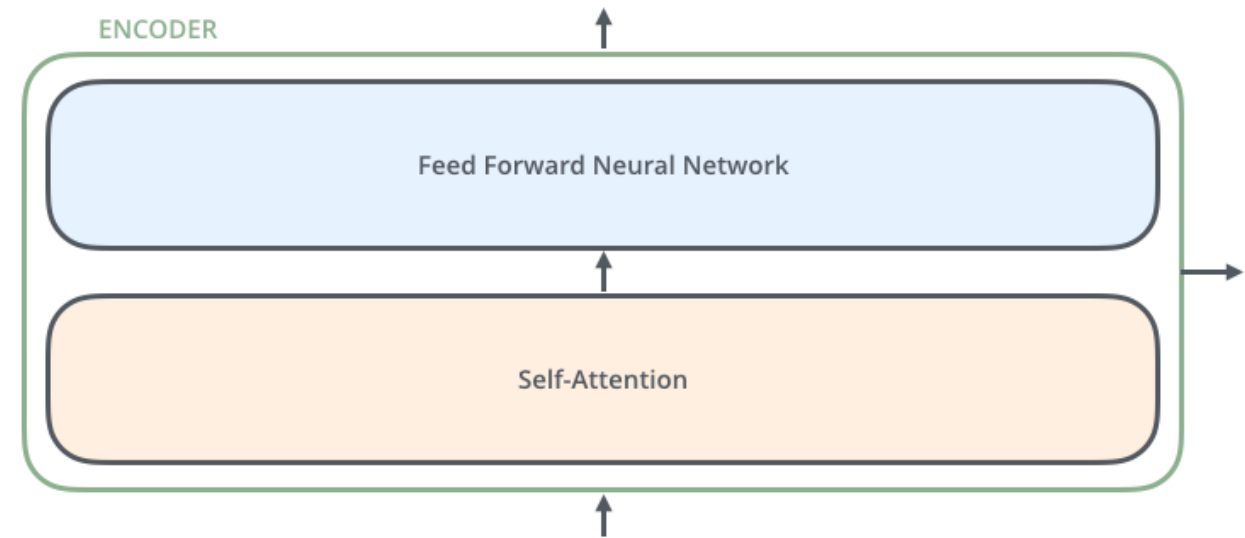
- Supongamos que la fase de encoder se compone de una **pila** de 6 **encoders**.
  - Este es el número usado en el paper original, pero puede ser cualquier otro número
- En el decoder tenemos una pila con el mismo número de unidades



<http://jalammar.github.io/illustrated-transformer/>

# Transformers

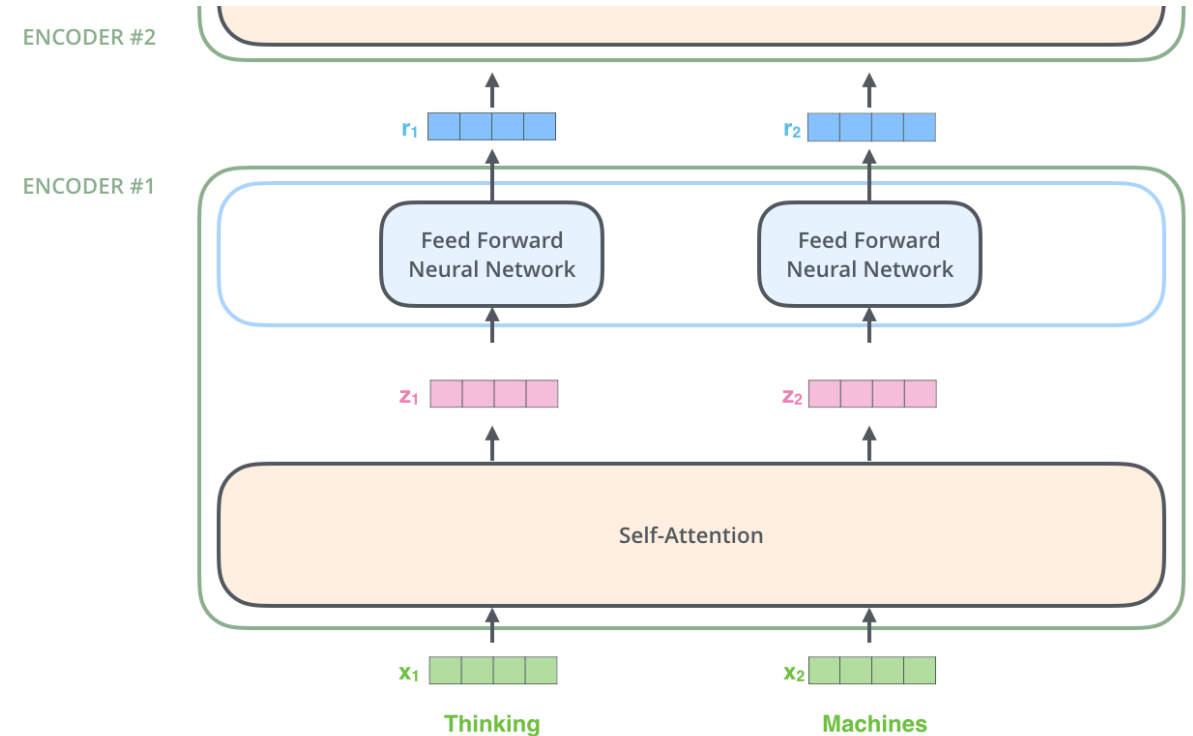
- Los **módulos del encoder** son iguales entre sí, y no comparten parámetros.
- Se dividen en **dos sub-capas**:
  - Una llamada **self-attention**: ayuda a codificar cada palabra “mirando” a las demás.
  - Una **capa feed-forward** (por ejemplo, perceptrón multicapa).





# Transformers

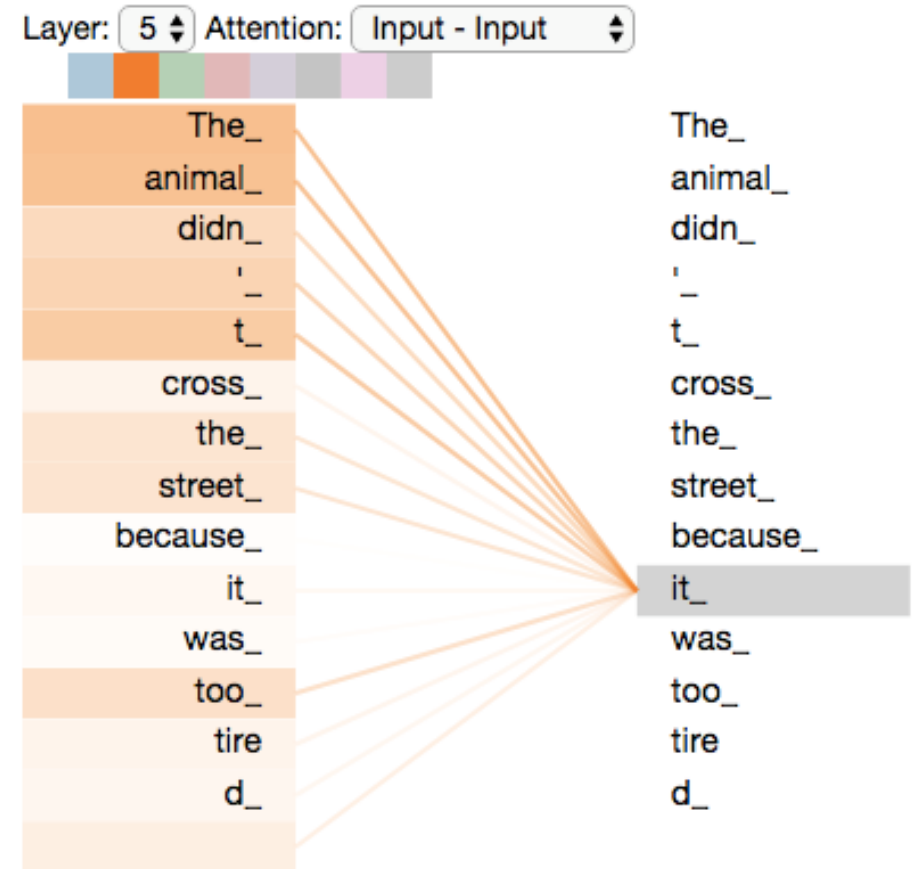
- En el primer módulo **encoder**:
  - Cada palabra de la secuencia de **entrada** se introduce usando una codificación con word **embedding** (**vector x**, en verde).
  - Después de self-attention, para cada palabra se crea un **vector z** (en rosa).
  - Después de la capa feed-forward, para cada palabra se devuelve un **vector r** (en azul), que será la entrada para el siguiente encoder.



# Transformers

- Capa **self-attention**:

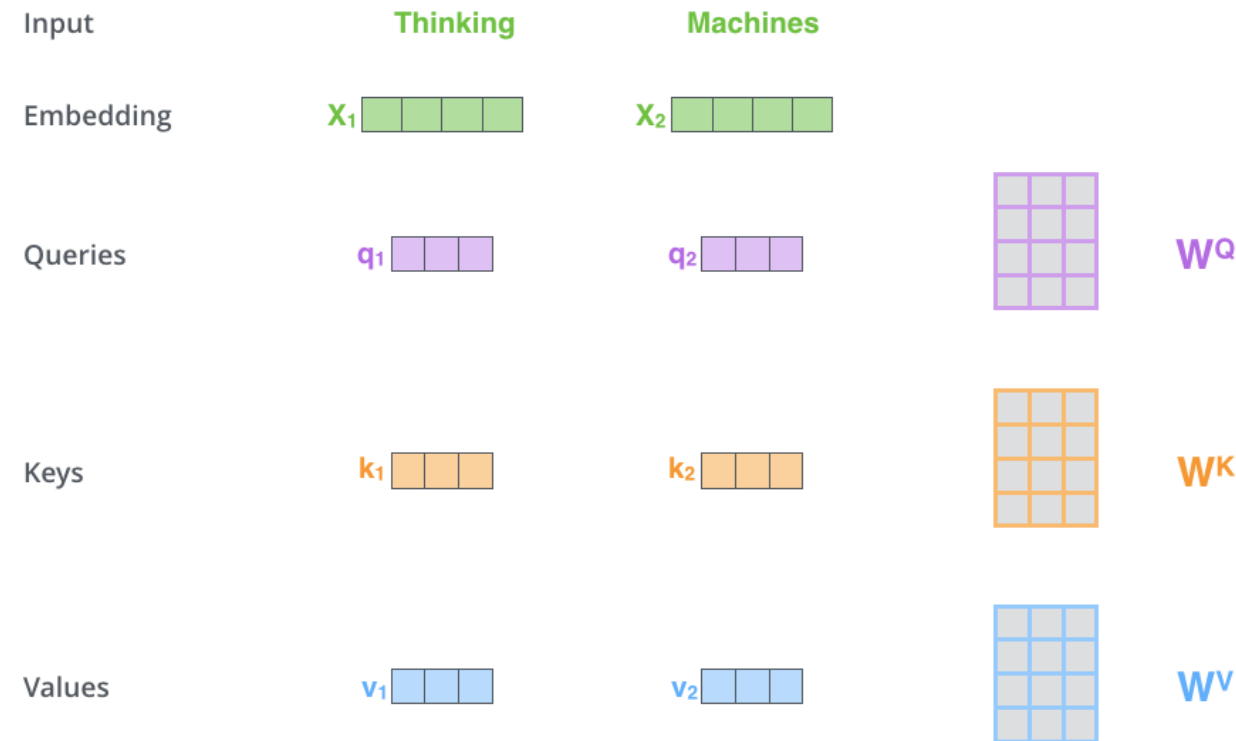
- Ayuda a relacionar cada elemento de la entrada con el resto.
- En el ejemplo, la capa asocia “it” a “Animal”, más que con el resto de palabras.



<http://jalammar.github.io/illustrated-transformer/>

# Transformers

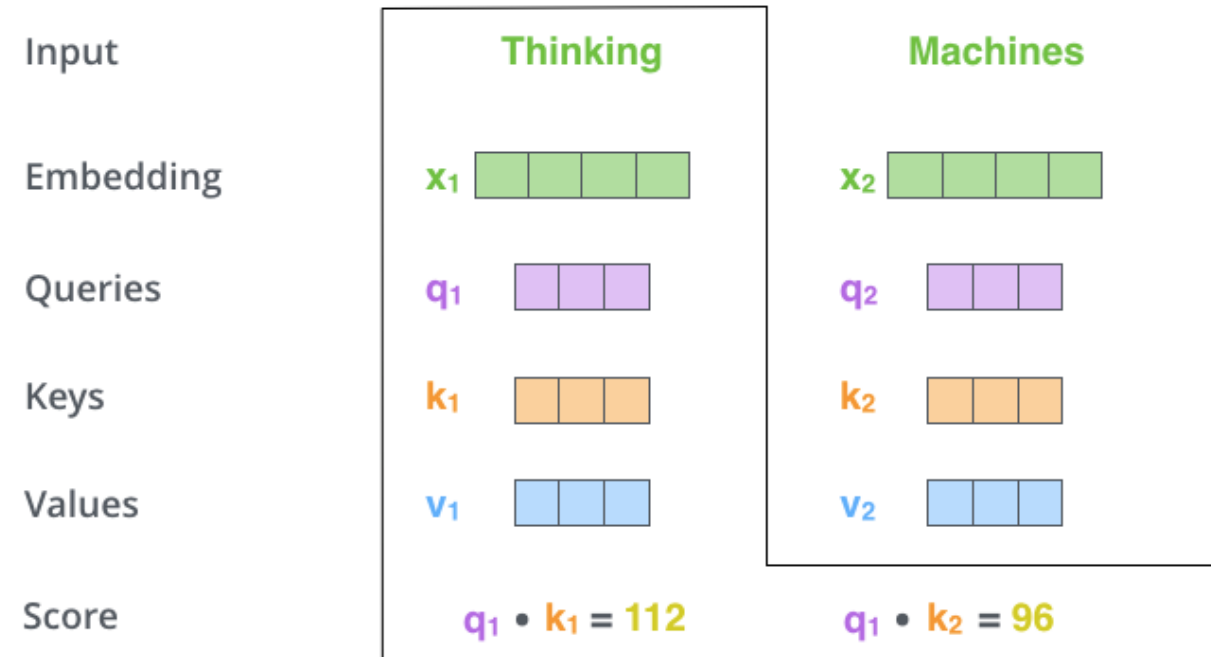
- Capa **self-attention**, hace uso de tres matrices (sus valores son parámetros de la red y se aprenden durante el entrenamiento):
  - Matriz de **Queries** ( $W^q$ )
  - Matriz de **Keys** ( $W^k$ )
  - Matriz de **Values** ( $W^v$ )
- Self-attention, **primer paso**:
  - Para cada palabra de entrada obtenemos **tres nuevos vectores: query ( $q$ ), key ( $k$ ) y value ( $v$ )**,
  - Se obtienen multiplicando cada palabra (vector  $x$ ) por cada matriz  $W^q$ ,  $W^k$  y  $W^v$ .



<http://jalammar.github.io/illustrated-transformer/>

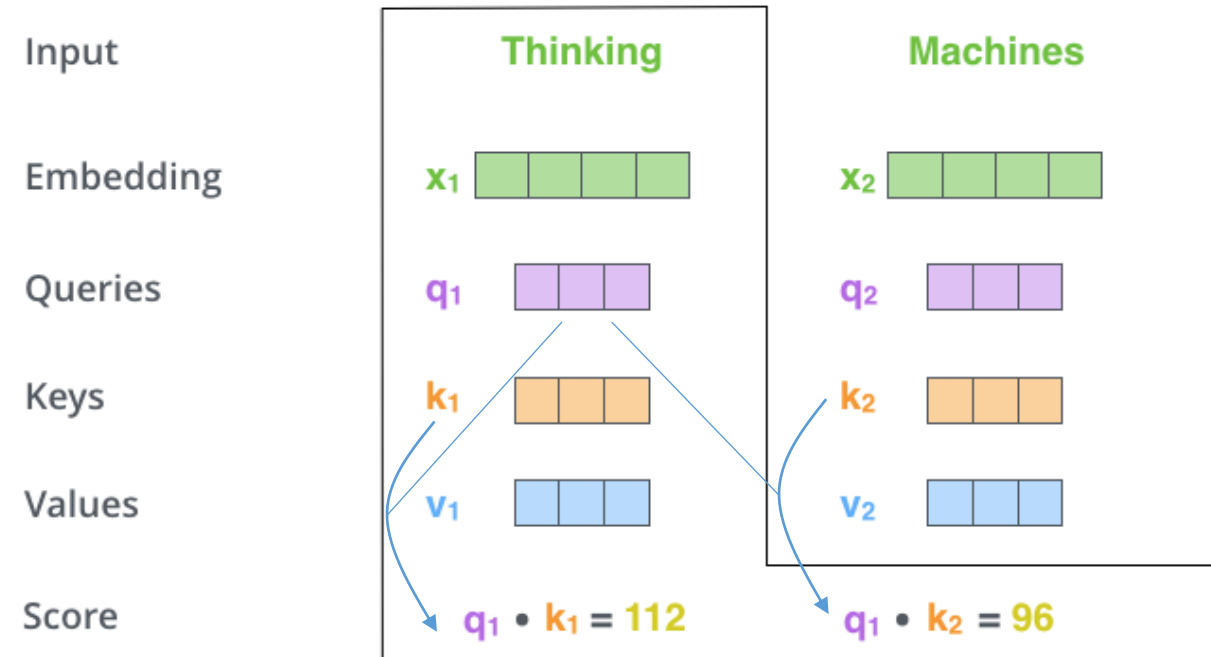
# Transformers

- Self-attention, **segundo paso**:
  - Para cada palabra, **puntuamos** el resto de palabras de entrada sobre ésta.
  - Para ello, se calcula el **producto escalar** de:
    - El vector **query** de la palabra referencia
    - El vector **key** de cada palabra del resto
  - El ejemplo solo muestra el cálculo para la primera palabra, pero esto también se hace con las demás.



# Transformers

- Self-attention, **segundo paso**:
  - Para cada palabra, **puntuamos** el resto de palabras de entrada sobre ésta.
  - Para ello, se calcula el **producto** escalar de:
    - El vector **query** de la palabra referencia
    - El vector **key** de cada palabra del resto
  - El ejemplo solo muestra el cálculo para la primera palabra, pero esto también se hace con las demás.



# Transformers

- Self-attention, **resto de pasos:**

Input

Embedding

Queries

Keys

Values

Score

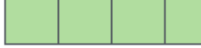
Divide by 8 (  $\sqrt{d_k}$  )

Softmax

Softmax  
X  
Value

Sum

Thinking

$x_1$  

$q_1$  

$k_1$  

$v_1$  

$q_1 \cdot k_1 = 112$

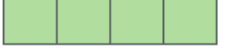
14

0.88

$v_1$  

$z_1$  

Machines

$x_2$  

$q_2$  

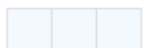
$k_2$  

$v_2$  

$q_1 \cdot k_2 = 96$

12

0.12

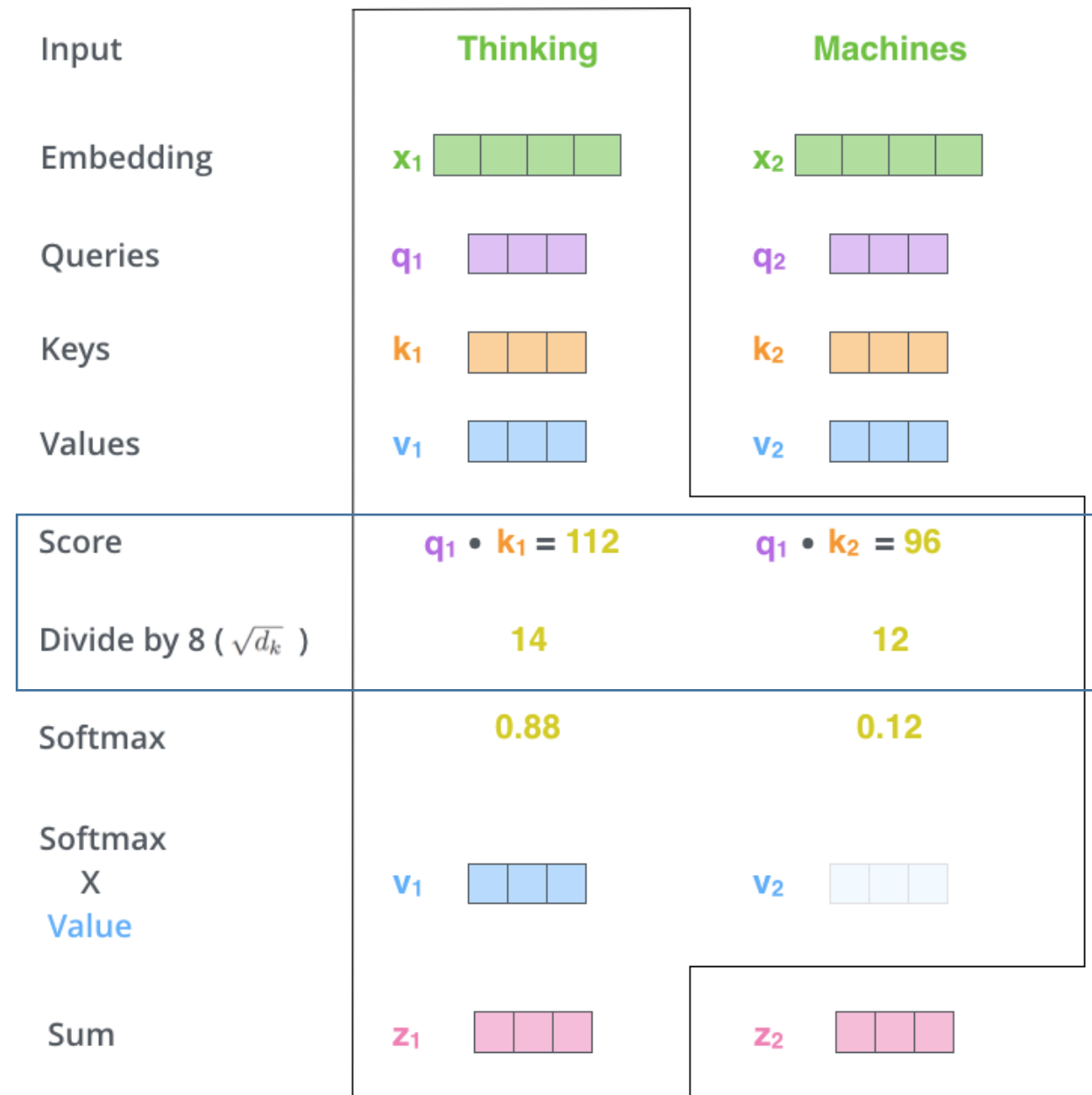
$v_2$  

$z_2$  

<http://jalammar.github.io/illustrated-transformer/>

# Transformers

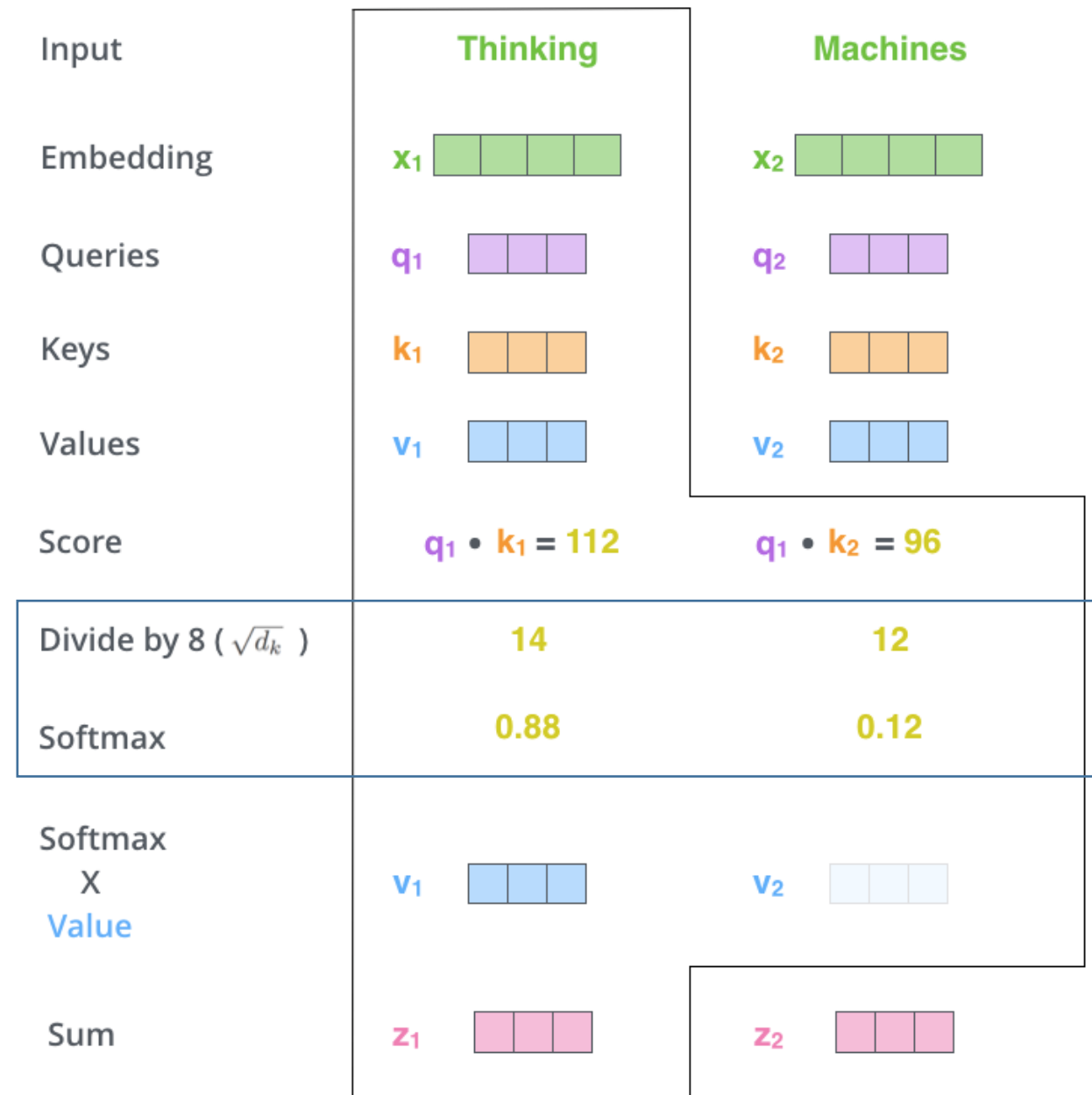
- Self-attention, **resto de pasos:**
  - **3:** Se **divide** cada **puntuación** por la raíz cuadrada de la dimensión del vector key (p.ej. se divide por 8 si el tamaño del vector  $k_i$  es 64)
    - Esto hace que los gradientes sean más estables



<http://jalammar.github.io/illustrated-transformer/>

# Transformers

- Self-attention, **resto de pasos:**
  - **3:** Se **divide** cada **puntuación** por la raíz cuadrada de la dimensión del vector key (p.ej. se divide por 8 si el tamaño del vector  $k_i$  es 64)
    - Esto hace que los gradientes sean más estables
  - **4:** El resultado se pasa por **softmax**

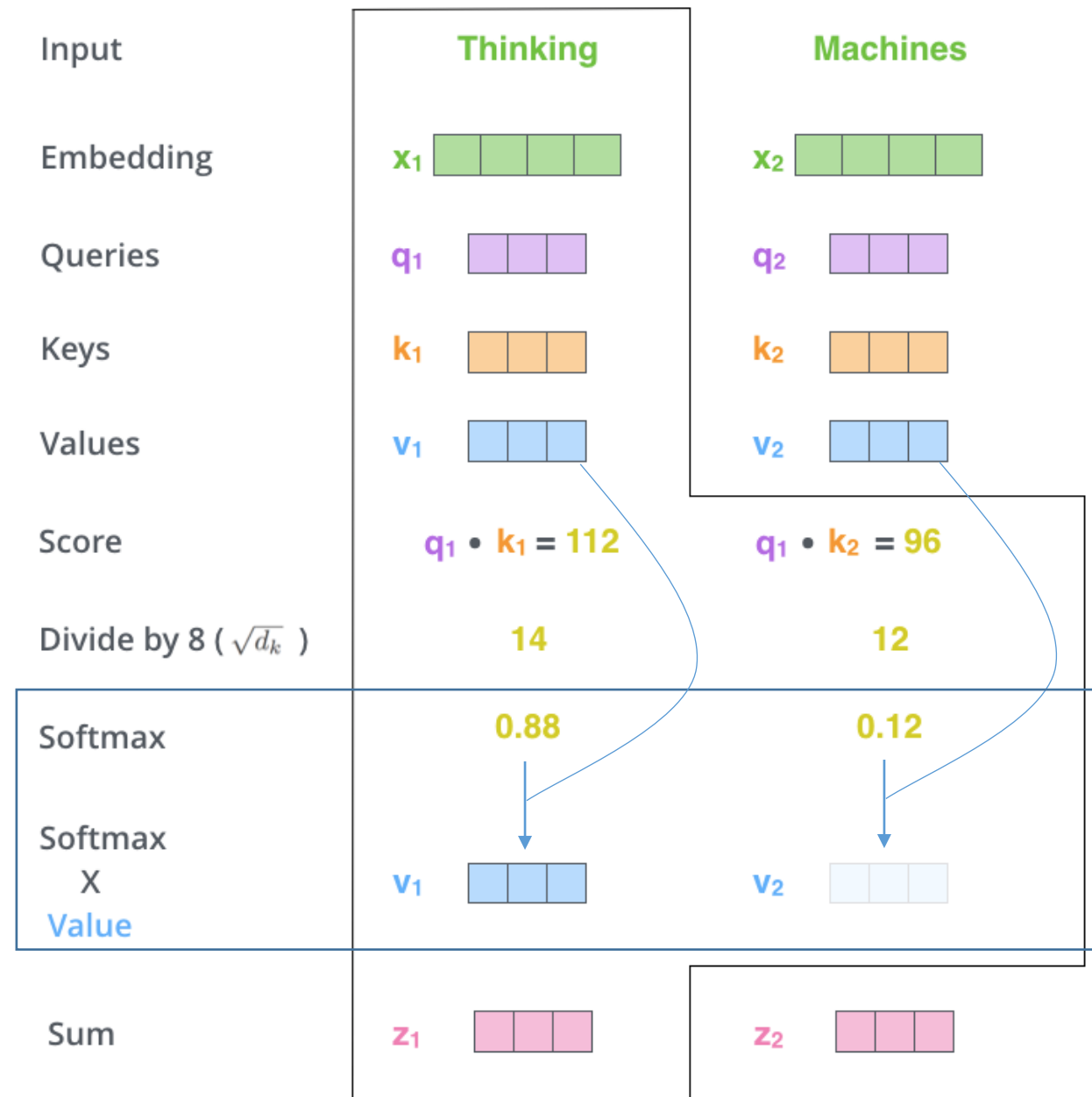


<http://jalammar.github.io/illustrated-transformer/>



# Transformers

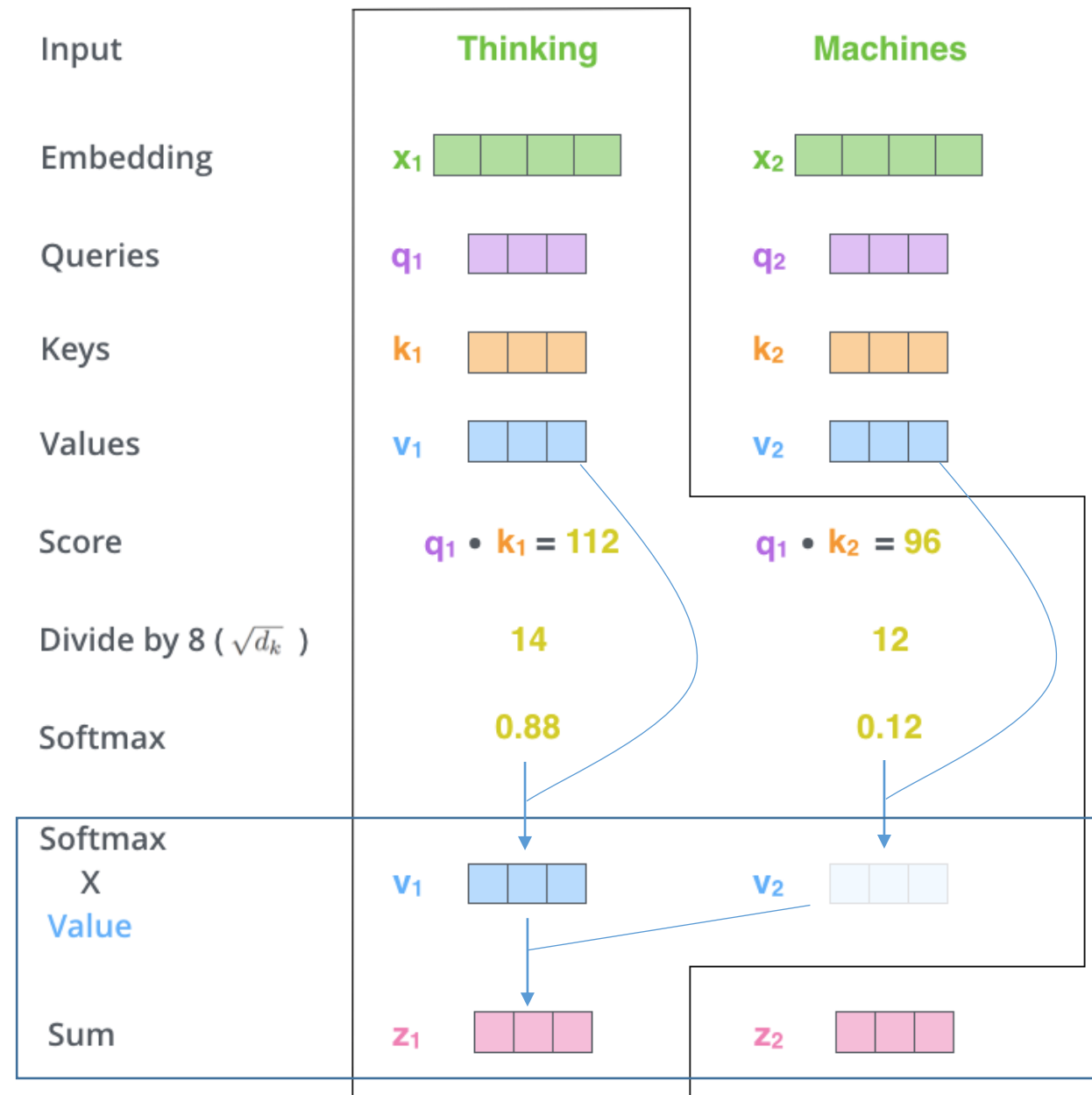
- Self-attention, **resto de pasos:**
  - **3:** Se **divide** cada **puntuación** por la raíz cuadrada de la dimensión del vector key (p.ej. se divide por 8 si el tamaño del vector  $k_i$  es 64)
    - Esto hace que los gradientes sean más estables
  - **4:** El resultado se pasa por **softmax**
  - **5:** Se **multiplica** el **vector Value** de cada palabra por cada valor correspondiente después del softmax



<http://jalammar.github.io/illustrated-transformer/>

# Transformers

- Self-attention, **resto de pasos**:
  - **3**: Se **divide** cada **puntuación** por la raíz cuadrada de la dimensión del vector key (p.ej. se divide por 8 si el tamaño del vector  $k_i$  es 64)
    - Esto hace que los gradientes sean más estables
  - **4**: El resultado se pasa por **softmax**
  - **5**: Se **multiplica** el **vector Value** de cada palabra por cada valor correspondiente después del softmax
  - **6**: Se **suman** los vectores resultantes en uno solo, el **vector z** (resultado)



<http://jalammar.github.io/illustrated-transformer/>

# Transformers

- Self-attention, **resto de pasos**:
  - **3**: Se **divide** cada **puntuación** por la raíz cuadrada de la dimensión del vector key (p.ej. se divide por 8 si el tamaño del vector  $k_i$  es 64)
    - Esto hace que los gradientes sean más estables
  - **4**: El resultado se pasa por **softmax**
  - **5**: Se **multiplica** el **vector Value** de cada palabra por cada valor correspondiente después del softmax
  - **6**: Se **suman** los vectores resultantes en uno solo, el **vector z** (resultado)

<http://jalammar.github.io/illustrated-transformer/>

Input

Embedding

Queries

Keys

Values

Score

Divide by 8 ( $\sqrt{d_k}$ )

Softmax

Softmax  
X  
Value

Sum

Thinking

$x_1$

$q_1$

$k_1$

$v_1$

$q_1 \cdot k_1 = 112$

14

0.88

$v_1$

$z_1$

Machines

$x_2$

$q_2$

$k_2$

$v_2$

$q_1 \cdot k_2 = 96$

12

0.12

$v_2$

$z_2$

# Transformers

- Todos los pasos en la capa **self-attention** que acabamos de ver, también se pueden implementar usando **operaciones matriciales** (álgebra lineal).
- Esto ayuda a conseguir un nivel mayor de paralelismo (las GPUs son buenas en este tipo de operaciones).

$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{Q}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{Q} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{K}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{K} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

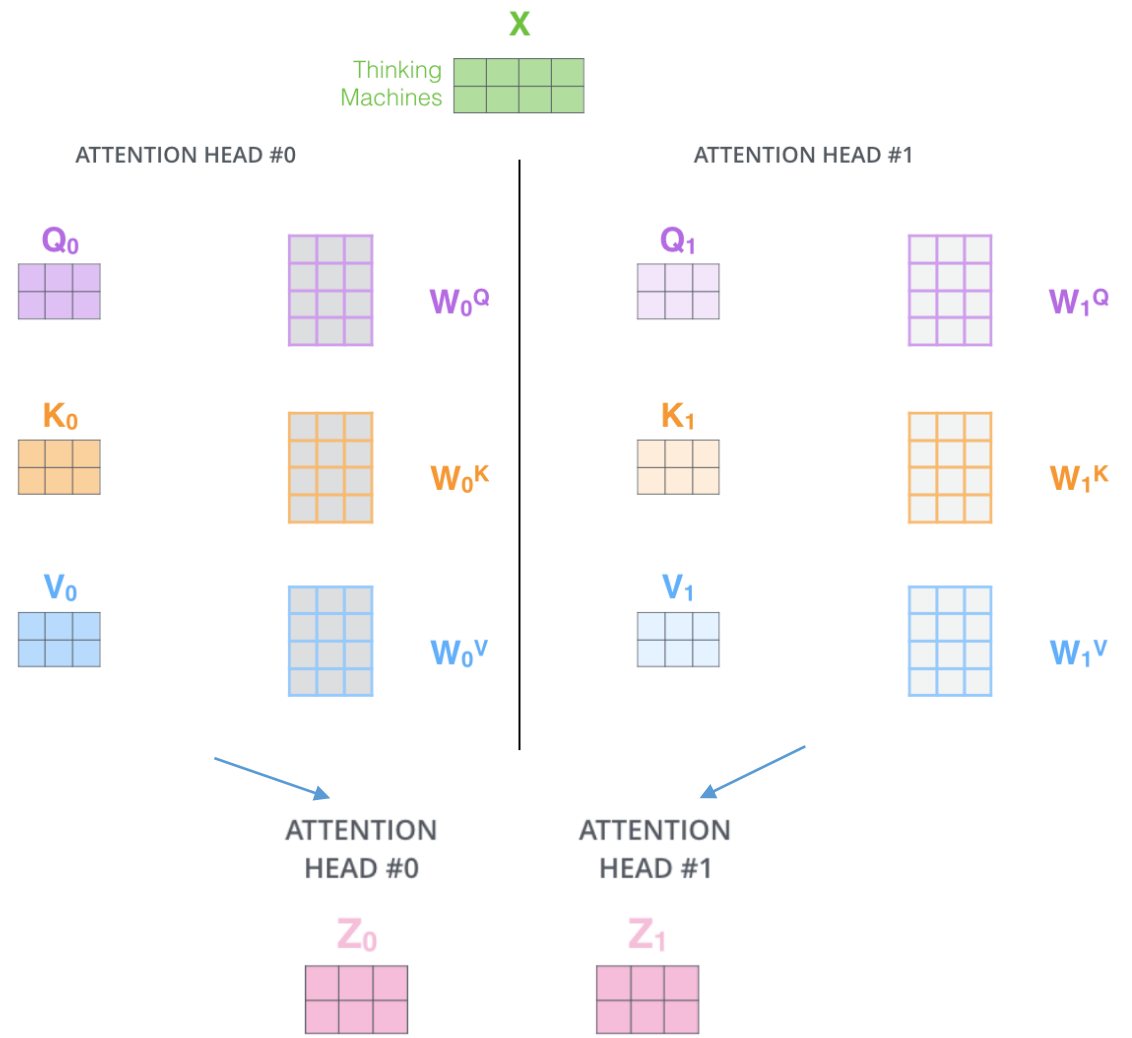
$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{V}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{V} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

$$\text{softmax} \left( \frac{\begin{matrix} \text{Q} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{K}^{\text{T}} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}}{\sqrt{d_k}} \right) \begin{matrix} \text{V} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$
$$= \begin{matrix} \text{Z} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

<http://jalammar.github.io/illustrated-transformer/>

# Transformers

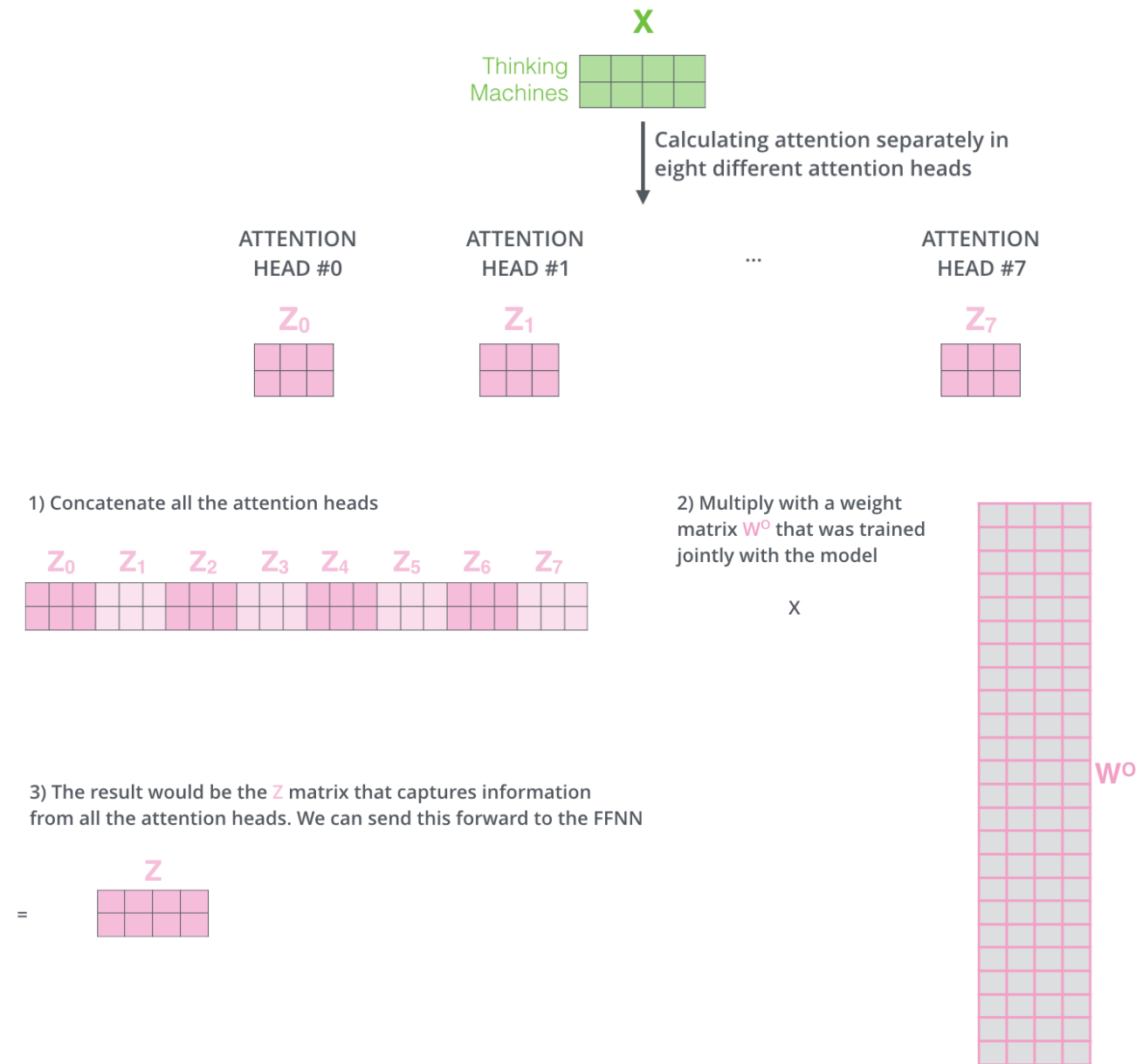
- En realidad, usamos una capa **multi-headed self-attention**, donde se repite la operación anterior varias veces en paralelo (con distintas “cabezas”).
  - Hay una matriz de Query/Key/Value por cabeza.
- Esto **ayuda** a:
  - Expandir la capacidad del modelo para centrarse en distintas posiciones.
  - Da la capacidad de tener múltiples representaciones del sub-espacio.



<http://jalammar.github.io/illustrated-transformer/>

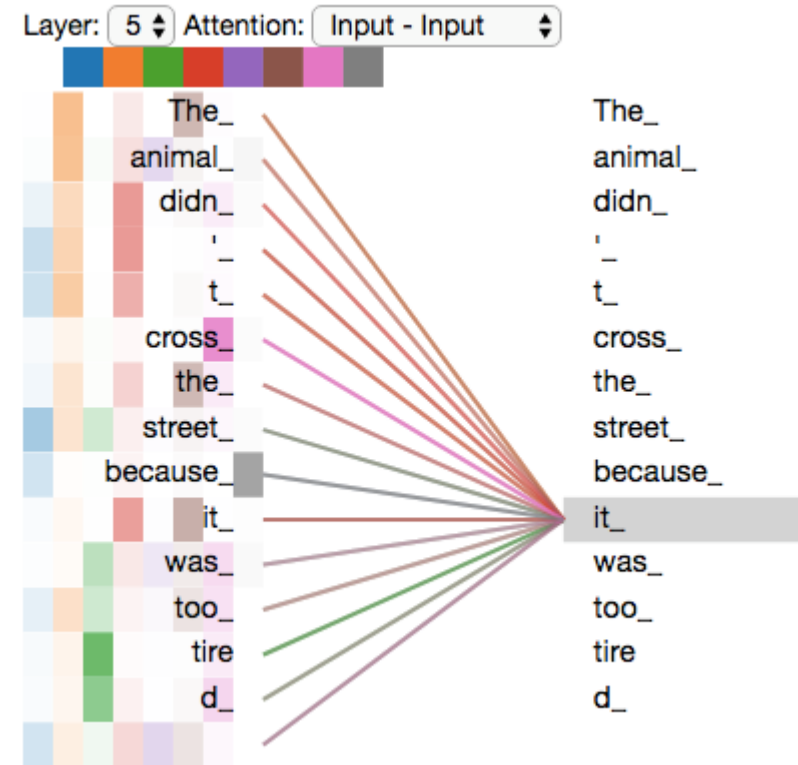
# Transformers

- Capa **multi-headed self-attention**:
  - A fin de tener un vector  $z$  por cada palabra (o lo que es lo mismo, una sola matriz  $z$ ), se **concatenan de forma ponderada** los resultados de cada cabeza
  - Se concatenan las matrices  $z$ , y se multiplican por una matriz  $W^0$ .



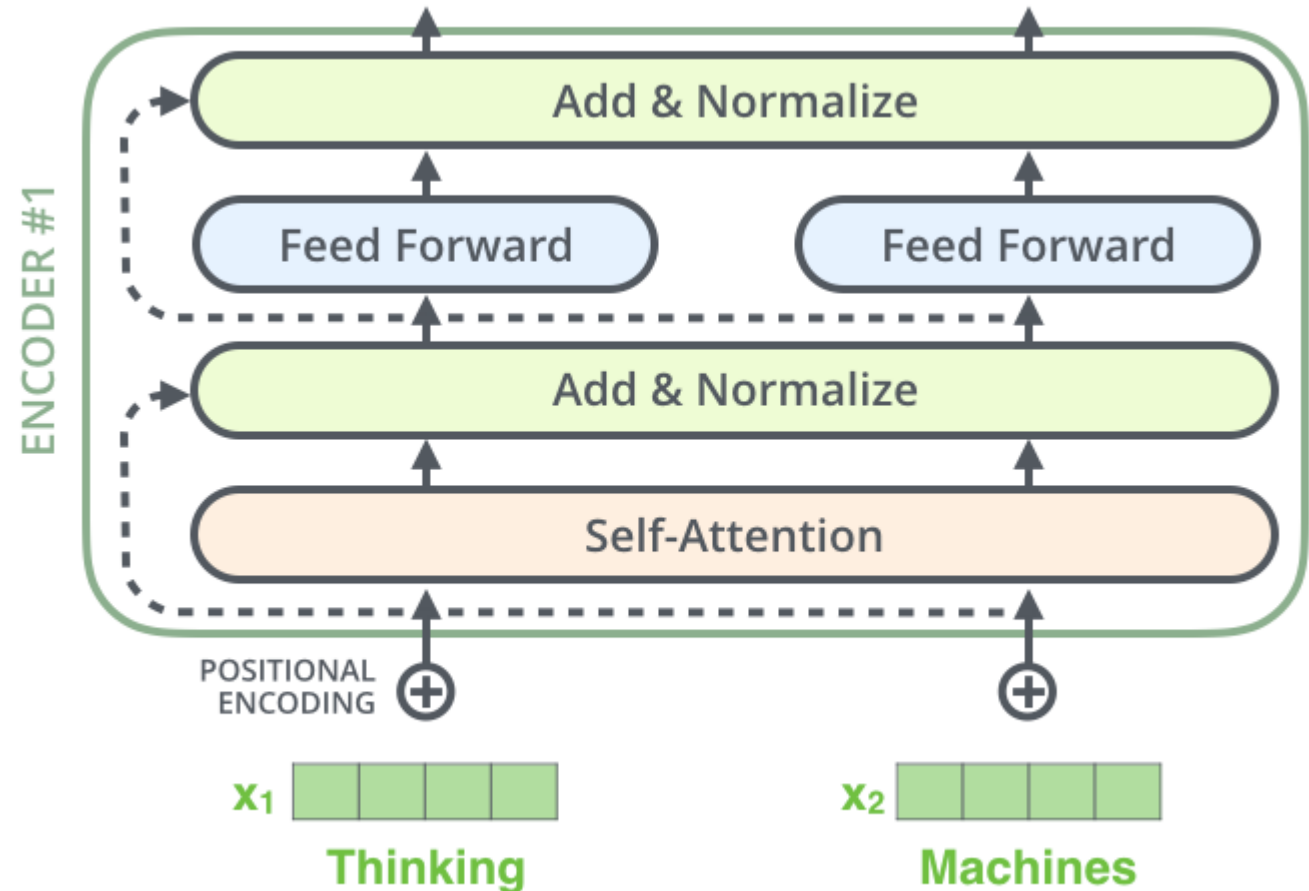
# Transformers

- Capa **multi-headed self-attention**:
  - El resultado es que el modelo es capaz de hacer distintas asociaciones de cada palabra con el resto.
  - Ver [Tensor2Tensor](http://jalammar.github.io/illustrated-transformer/)



# Transformers

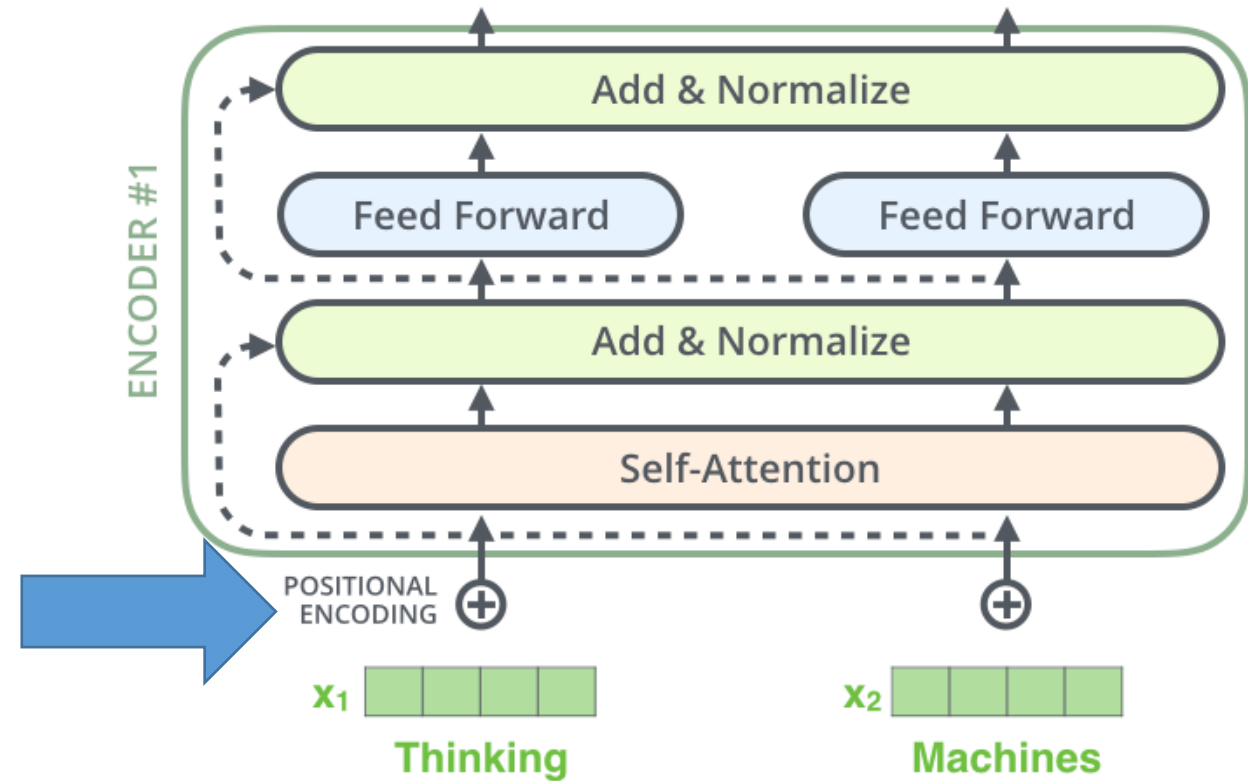
- Ya hemos visto cómo funciona la capa de self-attention, veamos la imagen completa de un encoder
- Vemos que los resultados de cada capa se **normalizan**
- Además hay conexiones **residuales** (en línea de puntos), con la misma idea que los módulos ResNet (ayudar a propagar gradientes).





# Transformers

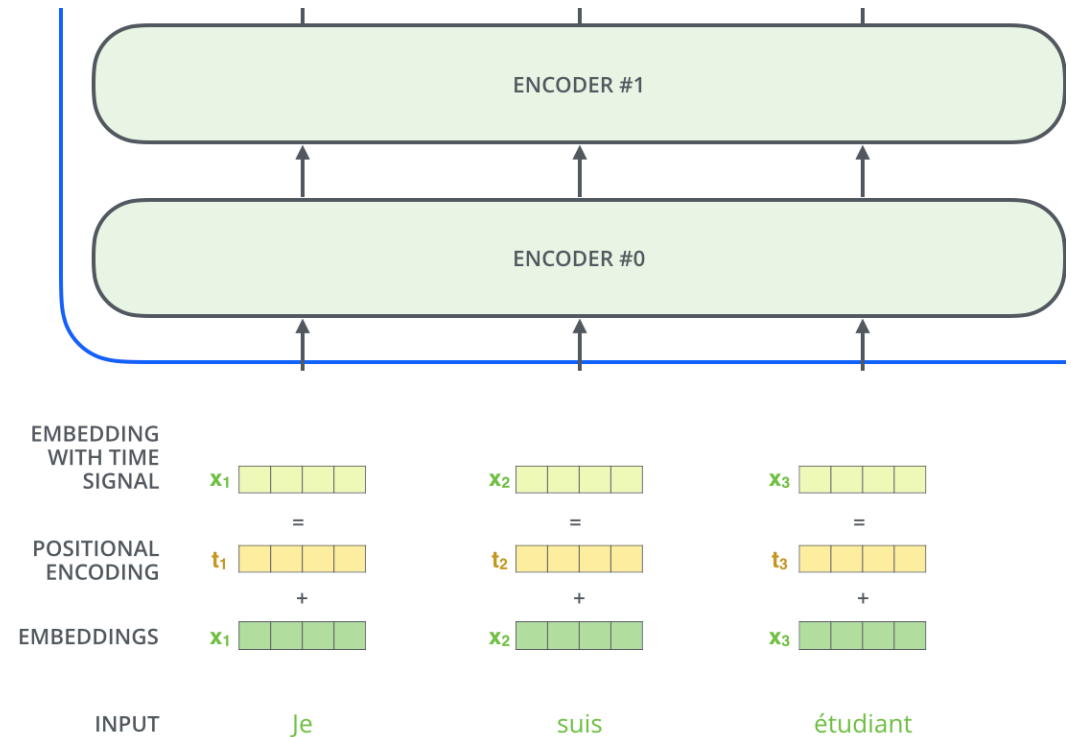
- El primer encoder recibe los elementos de entrada.
- Vemos que hay otra operación que se hace a los embeddings de cada palabra: **codificación posicional**.
- Piensa que en self-attention no tenemos información de qué posición ocupa cada palabra:
  - La codificación posicional ayudará a introducir esta información, modificando los valores de entrada.



# Transformers

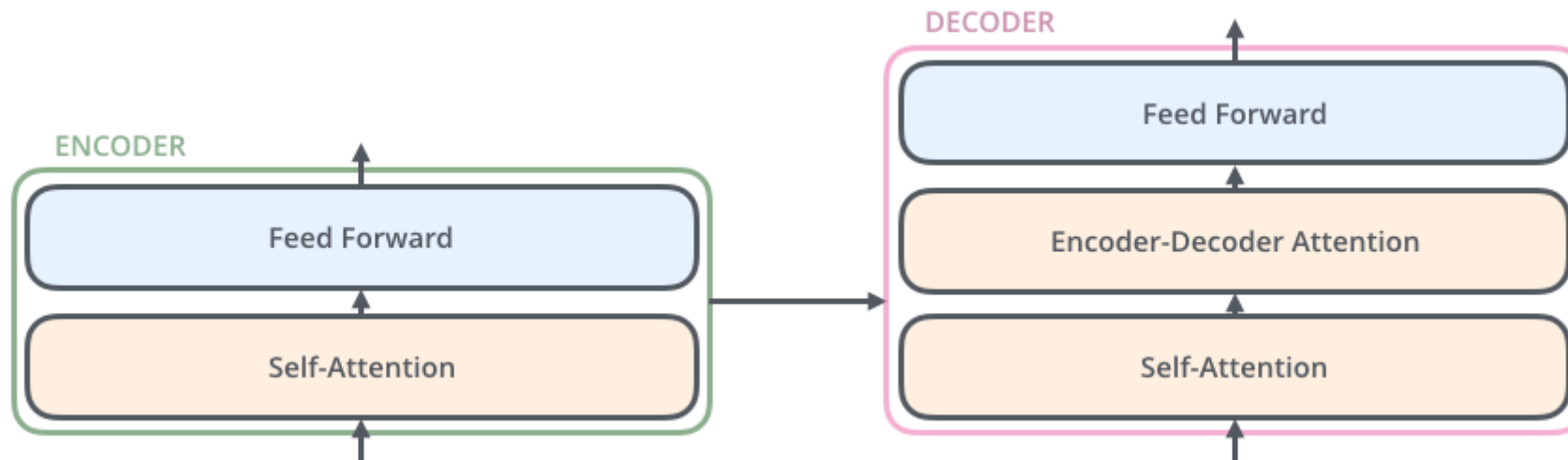
- **Codificación posicional:**

- Se suma cada embedding de entrada por un vector  $t$  (se aprende durante el entrenamiento).
- Se codifica una distancia por palabra, permitiendo jugar con distintas distancias (no siempre equidistantes).



# Transformers

- Los **módulos en el decoder** tienen también esas capas, pero entre ellas hay una capa especial de **Encoder-Decoder attention**, que ayuda al decoder enfocarse en partes relevantes de la entrada (como el sistema de attention que vimos en seq2seq).

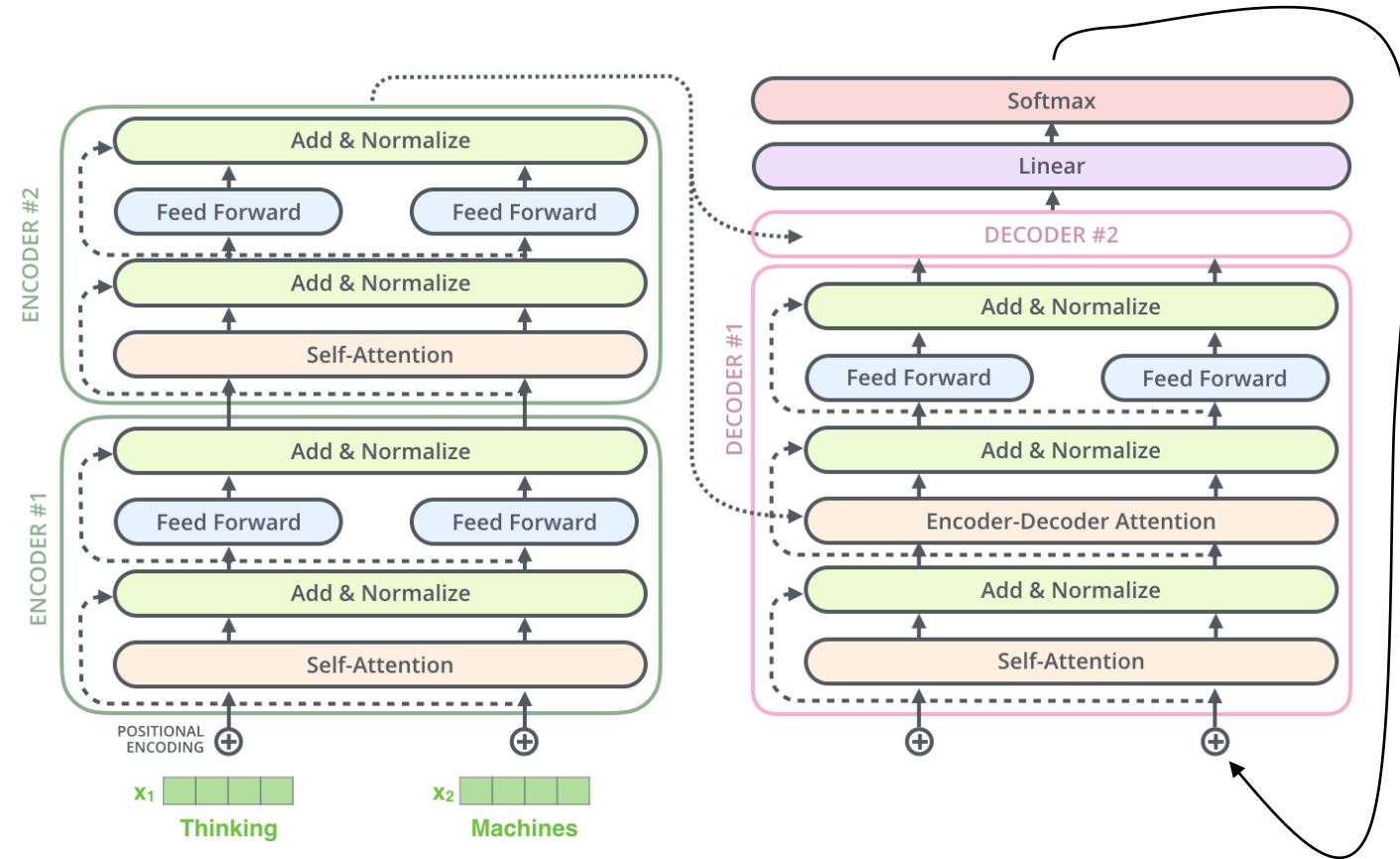


<http://jalammar.github.io/illustrated-transformer/>

# Transformers

- **El decoder:**

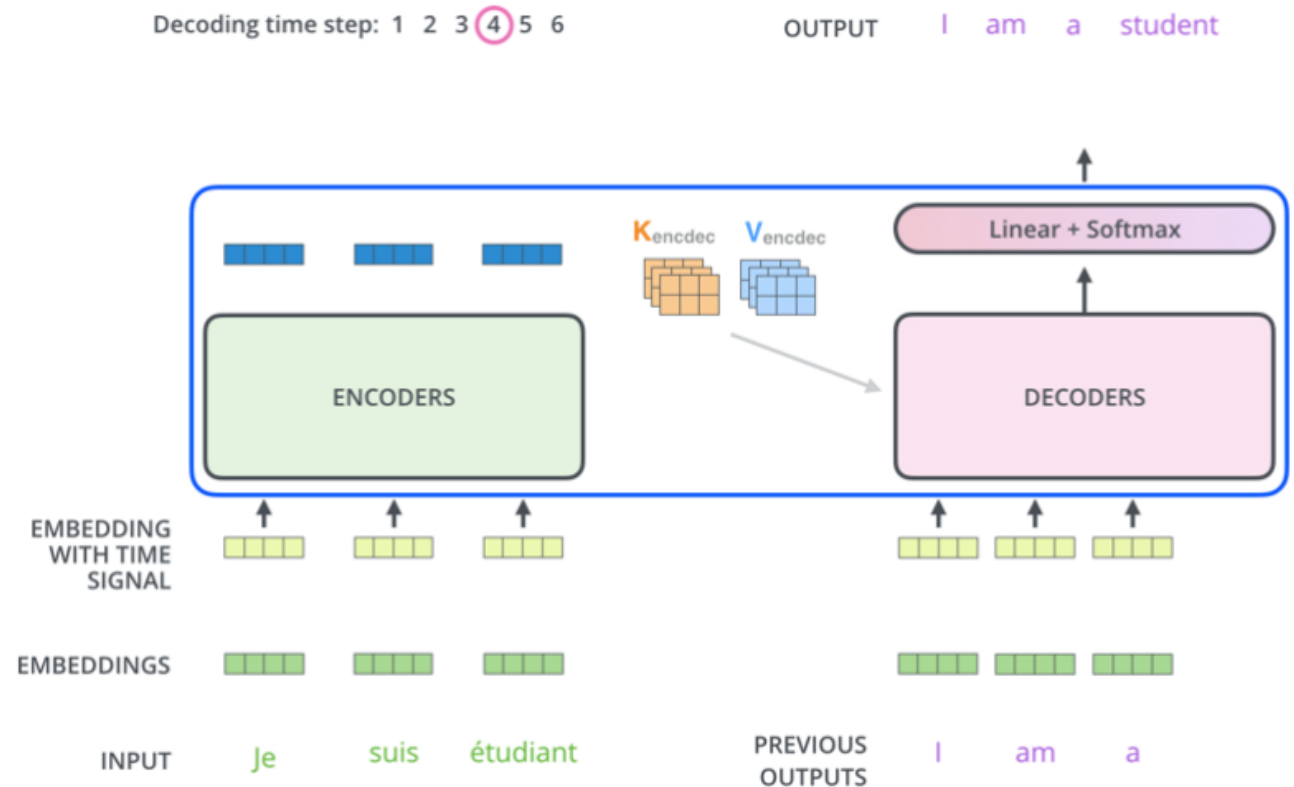
- Trabaja por **pasos**, generando una palabra (token) en cada paso.
  - Hasta generar un token de parada.
- Recibe **dos entradas**:
  - La **salida del último encoder** (va directo a cada módulo decoder).
  - La **salida del decoder en el paso anterior** (va al primer módulo del decoder).
- La **salida** es una softmax que indica el índice de la palabra (token) generado en el paso.



# Transformers

- **El decoder:**

- Las salidas del **encoder** se transforman en **vectores Key** y **Value** para las capas Encoder-Decoder Attention.
  - Se multiplican por dos matrices:  $K_{encdec}$ ,  $V_{encdec}$  (parámetros del modelo).
- Cada **token** generado se pasa como **entrada de nuevo** al decoder.
  - Se usa **codificación posicional**.
  - Aquellas entradas aún no generadas se multiplican por  $-\infty$ .



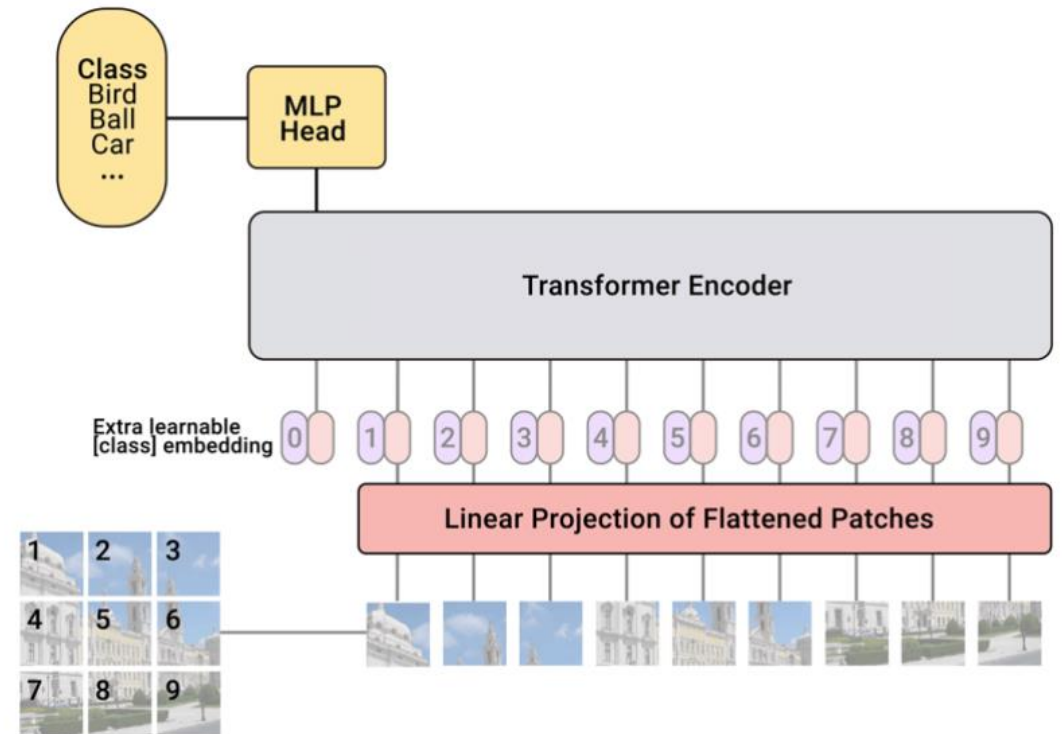
# Transformers

- Se han usado para los mejores modelos de lenguajes:
  - **BERT** (Google, 2018): sistema bidireccional, entrenado de forma no supervisada para predecir siguiente frase, 340 millones de parámetros.
  - **GPT** (OpenAI): entrenado de forma no supervisada para predecir siguiente palabra.
    - [Versión 1](#) (GPT, 2018):
      - Corpus de entrenamiento (texto) de 5GB, fine-tuning con pequeños datasets para tareas específicas.
      - 117 millones de parámetros 37 capas, secuencias de hasta 512 tokens, 1 mes de entrenamiento sobre 8 GPUs
    - [Versión 2](#) (GPT-2, 2019):
      - Corpus de 40GB (8 millones de webs), 1542 millones de parámetros, muchas GPUs, sin fine-tuning
      - “Modelo no publicado por precaución”...
      - Puedes probarlo en Talk to Transformer: <https://app.inferkit.com/demo>
    - [Versión 3](#) (GPT-3, 2020):
      - Corpus de 570GB (Wikipedia, libros, etc.), 175000 millones de parámetros (distribuido en muchas GPUs V100). Sin uso de fine-tuning.

# Transformers

- Otros usos además de lenguaje natural:

- [Imágenes](#)
- [Vídeo](#)
- [Música...](#)



# Recapitulando

- Las redes **RNN** (así como sus variantes LSTM y GRU) tienen problemas de eficiencia y falta de capacidad para modelar ciertas secuencias.
- El mecanismo de atención ayuda a paliar estos problemas, asociando elementos de la salida con varios de la entrada de forma explícita.
- Seq2seq es un modelo para generar secuencias a partir de secuencias, y contiene un encoder y un decoder.
- Los Transformers usan varios encoders y decoders usando self-attention, position coding, etc.
- Los Transformers están teniendo muchos usos, como en modelos de lenguaje natural (por ejemplo, GPT-3).